PHP: Crie um framework de acesso a dados usando Design Patterns

Clube Delphi +PHP

DevMedia group

Boas Práticas
Crie um agendador de tarefas usando técnicas de POO e Padrões de Projeto

Quick Update
Profiles no Delphi Prism e ASP.NET - Gerenciamento de Estado

API
Domine recursos avançados de desenvolvimento com a API do Windows

Ano 9 :: Edição 116 :: R$ 11,90

ISSN 1517990-7

Delphi como linguagem de Script

Permita que o usuário crie scripts e torne sua aplicação mais dinâmica

Brinde

Comprando esta edição ganhe 4 vídeo-aulas:

Documentação básica para desenvolvimento de sistemas:

1 – Artefatos
2 - Entenda a importância da documentação
3 – Levantamento de requisitos
4 – Diagramas UML

Cursos Online

Web Mobile PLUS

A **Revista WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade **.Conheça abaixo os cursos já disponíveis.**

Curso de .net em destaque

Aplicação completa de orçamento Doméstico no Visual Studio 2005

Confira neste curso online como criar uma aplicação completa no Visual Studio 2005, usando ASP.NET, Web Services, Mobile e muito mais! Veja como criar uma aplicação de orçamento doméstico, usando diagrama de classes de uma forma muita produtiva, criar classes de negócios de acesso a dados.

Confira o plano de aula completo:
www.devmedia.com.br/domesticovs

Curso de Java em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **WebMobile** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Em breve mais novidades para você! www.devmedia.com.br/curso

DevMedia GROUP **Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025**

Sumário

Quick Update

6 – Profiles no Delphi Prism e ASP.NET
[*Guinther Pauli*]

Delphi

Boa Ideia

8 – Metaprogramação
Interpretando código com o Delphi
[*Vitor Luiz Rubio*]

Mão na Massa

20 – Programação com a API do Windows
Dicas gerais para tarefas comuns
[*Vitor Luiz Rubio*]

Boas Práticas

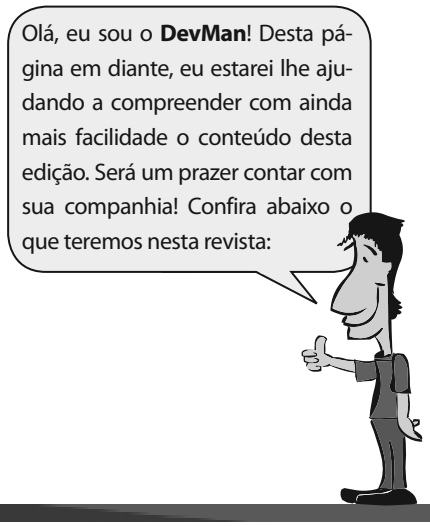
Boas Práticas

37 – Crie um agendador de tarefas OO – Parte 1
Princípios de OO
[*Rafael Stavarengo*]

PHP

PHP

44 – Design Patterns em PHP
Entenda o padrão Abstract Factory construindo um framework de acesso a dados
[*Lucas Simões Maistro e Raphael Zanon Rodrigues*]



Brinde na web desta edição - 4 vídeos! Confira:

4 Vídeos

- **Documentação básica para desenvolvimento de sistemas: Parte 1 –** O autor apresenta os artefatos necessários e obrigatórios para criação de um sistema, utilizando UML para diagramação, levantamento de requisitos e DER
- **Documentação básica para desenvolvimento de sistemas: Parte 2 –** Entenda a importância da documentação
- **Documentação básica para desenvolvimento de sistemas: Parte 3 –** Entenda o que são requisitos e como realizar um bom levantamento dos mesmos
- **Documentação básica para desenvolvimento de sistemas: Parte 4 –** Tenha uma abordagem geral do que é UML e de quais diagramas ela é composta

Para acessar os vídeos utilize os seguintes dados
Login: ***DVM.PL*** *Senha:* ***DUAL***

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PL** Senha: **DUAL**

Clube Delphi +PHP

Ano 9 - 116ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.laporte@gmail.com

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Capa e Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Os principais softwares do mercado suportam scripts. Podemos definir aqui o termo script como a possibilidade de o usuário final do produto escrever alguma porção de código que adicione funcionalidade extra à aplicação, sem que para isso seja necessário recompilar todo o sistema ou acionar a equipe de desenvolvimento. Alguns exemplos: vários geradores de relatórios atuais permitem que o usuário final digite algum tipo de script que permite definir em runtime como dados serão apresentados, softwares comerciais permitem que o usuário defina expressões que são avaliadas por um engine em tempo de execução que podem definir regras de negócio dinamicamente, tudo altamente personalizado e de acordo com a necessidade do cliente. Pessoalmente já trabalhei em projetos onde o usuário podia criar seus próprios formulários de dados em Delphi, praticamente uma ferramenta case, onde ele também tinha a possibilidade de digitar código, usando Pascal Script. Isso eleva o poder de customização do seu software ao extremo, pode reduzir custos, usuários mais experientes podem definir comportamentos altamente específicos de acordo com o seu negócio, sem ocasionar manutenção extra. É um recurso muito interessante e que trazemos como destaque desta edição, na matéria do Vitor Rubio.

O mesmo autor ainda traz um artigo completo sobre as principais tarefas que o desenvolvedor precisa saber para desenvolver com a API e Registro do Windows. Customize o seu software ao máximo para tirar todos os proveitos oferecidos pelo sistema operacional, mesmo aqueles não suportados diretamente pela VCL do Delphi.

Boas Práticas - Como de costume, sempre trazemos o melhor da Programação Orientada a Objetos com o Delphi, além de vários outros temas relacionados. Na primeira parte do artigo do Rafael, você aprenderá a desenvolver um pequeno software utilizando algumas técnicas de POO, que mais adiante, serão otimizadas ao serem aplicados poderosos padrões de projeto.

Na sessão PHP também temos boas práticas, aprenda a aplicar poderosos padrões de projeto em Web Sites com PHP, no artigo do Lucas e do Raphael.

Boa leitura!

Guinther Pauli

guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia group

Perguntas e Respostas

Programação com Banco de Dados na Web – Que tecnologia escolher para acesso a dados no ASP.NET?

Perguntas respondidas por:
Guinther Pauli: guinther.pauli@gmail.com

Estou iniciando o desenvolvimento de aplicações Web com ASP.NET e Delphi Prism. Porém estou bastante confuso no que diz respeito ao acesso a dados. São muitas opções, o que você indica para eu utilizar em um pequeno projeto? E em um projeto de médio porte?

Eliseu Bassani
São Borja – RS

Olá Eliseu. A base de acesso a dados no .NET, seja o tipo de aplicação que você está desenvolvendo, é o ADO.NET. Então, é de fundamental importância que você conheça bem esta tecnologia. Ela possui inúmeras classes e interfaces, muito semelhante ao que temos hoje na VCL do Delphi, no dbExpress e no DataSnap. O problema é que normalmente você terá que criar o acesso a dados via código, diferentemente do que estamos acostumados no Delphi Win32. Isso se deve ao fato de que estamos em um ambiente Web, onde não podemos ter um "DataModule" que esteja instanciado inteiro em memória e fique lá residindo durante todo a sessão do usuário, isso afetaria a performance da aplicação. E na Web cada byte é precioso, cada milessegundo perdido é um desperdício.

Veja na **Listagem 1** um exemplo de código ADO.NET que insere dados de um formulário em um banco de dados.

Uma abordagem um pouco mais poderosa, e minha preferida, é separar todo o código ADO.NET em uma camada lógica à parte, o que chamamos de DAL – Data Access Layer (uma implementação do padrão DAO - Data Access Objects). Imagine isso como um "DataModule". Ao invés de passar os dados dos controles de tela diretamente para a DAL, instanciamos um objeto DTO – Data Transfer Object, por exemplo, um TCliente com os campos Nome, Endereço etc. A DAL recebe o DTO e usa ADO.NET para persistir no banco. Solução ideal para casos onde a empresa quer desenvolver uma solução sem se atrelar diretamente a uma tecnologia em específico, recomendada para médios e grandes projetos. Mais tarde, você pode facilmente adaptar a solução para outra tecnologia que venha ser lançada, visto que está tudo separado em camadas e existe uma fraca dependência (acoplamento) entre elas.

Uma outra excelente alternativa é você considerar o uso de um framework de persistência pronto (no caso anterior você desenvolve seu framework). O framework de mapeamento objeto relacional e persistência mais robusto e consolidado do mercado hoje que você pode usar com o Delphi Prism é o NHibernate (veja edição 115). Lembrando que a Microsoft também possui um framework com o mesmo propósito, chamado ADO.NET Entity Framework. Veja nesse caso que um ponto importante a ser considerado é a curva de aprendizado, pois essa abordagem difere totalmente da VCL Win32.

Listagem 1. Inserindo dados no banco via ADO.NET "puro"

```
var Con := new SqlConnection(ConStr);
var SQL := "insert into CLIENTES values(@NOME)";
var Cmd := new SqlCommand(SQL, Con);
Cmd.Parameters.AddWithValue("@NOME",txtNOME.Text);
Con.Open();
try
  Cmd.ExecuteNonQuery();
finally
  Con.Close();
end;
```

Para um desenvolvimento mais RAD e simples (pequenos projetos), mais semelhante com o que estamos acostumados com a VCL do Delphi, sugiro o uso de DataSources. Eles se conectam diretamente a controle de tela como GridView e DetailsView e reduzem drasticamente a quantidade de código necessário para criar aplicações de banco de dados para Web. Um ótimo exemplo é o SqlDataSource. Como todas as outras tecnologias ele também usa ADO.NET de fundo e faz uso de todos os seus recursos. Para mais informações, sugiro consultar o seguinte artigo em nosso site: *http://www.devmedia.com.br/articles/viewcomp.asp?comp=14013.* ●

Envie suas dúvidas para nossa equipe de especialistas: *guinther@devmedia.com.br/ pauloquicoli@devmedia.com.br/ daniel.laporte@devmedia.com.br/ rodrigo.carreiro@devmedia.com.br*

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Quick Update:
Profiles no Delphi Prism e ASP.NET

Guinther Pauli
guinther.pauli@gmail.com
http://gpauli.com
http://twitter.com/guintherpauli
Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Kylix, certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET). Editor da Revista ClubeDelphi e da Revista .NET Magazine Brasil, professor de Pós-Graduação da UNIVEM-SP. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Foi arquiteto e desenvolvedor do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

O ASP.NET possui inúmeros recursos para manter o estado em aplicações Web, onde podemos citar o simples uso de Cookies, passando por variáveis de Sessão, Cache, ViewState e mais. Imagine a seguinte situação: o usuário entra no seu site e customiza algumas partes da página, define algumas preferências de navegação, usabilidade etc. Quando ele retornar ao seu site, não importa quanto tempo isso leve, devemos ter uma forma de "lembrar" essas preferências definidas pelo usuário. Antigamente (ASP.NET 1.x), criávamos uma tabela no banco de dados com campos que definiam o perfil (profile) do usuário. Após a sua visita, gravávamos suas preferências no banco e o ID do usuário em um cookie local no browser. Quando ele retornava, recuperávamos as informações a partir do cookie.

Um grande recurso introduzido pelo ASP.NET 2.0 é o suporte a Profiles. Este recurso faz basicamente o que expus no parágrafo anterior, porém, reduzindo drasticamente o trabalho necessário para realizar esta tarefa. Com o Delphi Prism, podemos utilizar o recurso de Profiles em aplicações Web com ASP.NET 3.5 e Visual Studio 2008 (versão de produção atual no momento do fechamento desta edição).

Vamos a um exemplo simples. Vamos permitir que o usuário defina suas informações de perfil no site, quando ele retornar, vamos recuperar essas informações e exibir na tela. Inicie um novo ASP.NET Web Site no Delphi Prism (**Figura 1**). No formulário principal, coloque alguns controles TextBox na tela com um texto acima, como mostra a **Figura 2**. Coloque também um Button.

No Solution Explorer, abra o arquivo Web.Config e logo abaixo da abertura do elemento System.Web digite a definição de profile da **Listagem 1**.

Listagem 1. Defininindo o profile no Web.Config

```
<profile>
   <properties>
      <add name="NOME"/>
      <add name="EMAIL"/>
   </properties>
</profile>
```

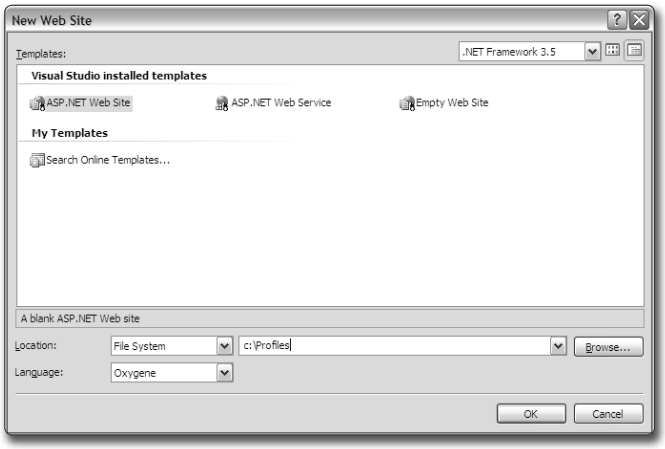


Figura 1. Criando um ASP.NET Web Site no Delphi Prism

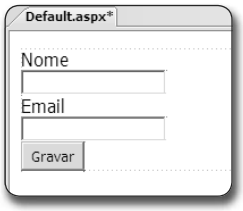


Figura 2. Formulário da aplicação

Dê um Build no Web Site, dê um duplo clique no botão gravar no form e digite o código da **Listagem 2**. Observe a "mágica" aqui. O Delphi Prism fez um parse no Web.Config, verificou o que colocamos na sessão Profile (campos) e criou automaticamente uma classe "wrapper" que é injetada na página através da propriedade Profile. Repare que ela é uma classe **fortemente tipada**. Por padrão, como não definimos os tipos nos campos no Web.Config, ambos são strings. Tudo o que fazemos aqui então é atribuir ao Profile os valores digitados nos TextBox's.

Listagem 2. Gravando dados no perfil

```
method _Default.Button1_Click(sender: System.Object; e:
 System.EventArgs);
begin
   Profile.NOME := TextBox1.Text;
   Profile.EMAIL := TextBox2.Text;
end;
```

No Page_Load da página (**Listagem 3**) fazemos exatamente o procedimento contrário. Pegamos os valores do Profile e realimentamos os controles de tela, para recuperar os dados, o estado do perfil.

Executando a aplicação (CTRL+F5), digitamos os valores nos campos da tela, clicamos no botão e fechamos o browser. Ao retornar, vemos que os dados ainda estão lá! Veja a **Figura 3.**

Vamos examinar o que aconteceu aqui. No momento que acessamos o site pela primeira vez, o ASP.NET acionou o recurso de Profiles. Com isso, foi criado "on-the-fly" um banco de dados prontinho com uma estrutura previamente definida para guardar os dados do perfil dos usuários. Quer ver? No Solution Explorer dê um refresh na pasta App_Data, veja que foi criado um database chamado ASPNETDB, explore sua estrutura e verifique as tabelas criadas, verifique também que os dados que acabamos de digitar também estão lá. De praxe, o ASP.NET já gravou um cookie no browser para mapear o usuário para o seu respectivo profile no servidor quando ele entrar novamente.

Figura 3. Dados persistem entre várias sessões do mesmo usuário

Listagem 3. Recuperando dados do Profile

```
method _Default.Page_Load(sender: Object; e: EventArgs);
begin
    if not IsPostBack then
    begin
        TextBox1.Text := Profile.NOME;
        TextBox2.Text := Profile.EMAIL;
    end;
end;
```

Algumas considerações sobre o recurso:

- Por padrão o recurso usa um banco pré-definido no SQL Server chamado ASPNETDB, mas você pode criar a estrutura em um banco SQL Server existente;
- Com um pouco mais de trabalho é possível usar outros bancos de dados;
- É possível gravar vários tipos de informações no banco que não só strings, mas coleções completas (por exemplo um carrinho de compras do usuário) bem como tipos definidos pelo próprio desenvolvedor. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Metaprogramação

Interpretando código com o Delphi

Do que trata o artigo

Metaprogramação pode ter vários significados ou focos. Neste artigo estudaremos alguns componentes para interpretar código pascal. A ideia é criar uma maneira de disponibilizar ao usuário uma interface onde ele possa "programar" suas próprias rotinas e que estas possam até interagir com elementos do nosso software.

Para que serve

Na perspectiva do desenvolvedor, a metaprogramação serviria para economizar trabalho e linhas de código, tornar o sistema maleável ou mutável e, caso se use a metaprogramação para gerar código, este seria menos sujeito a erros porque foi gerado automaticamente. Do ponto de vista do usuário a metaprogramação significa uma maneira de economizar dinheiro, pois dependendo da alteração que ele precise no sistema ele mesmo pode fazê-la.

Em que situação o tema é útil

Este tema é útil em situações onde o custo para se alterar um software especificamente para uma empresa é alto demais tanto para o desenvolvedor como para a empresa, e ainda tornaria o sistema específico e invendável para uma empresa de ramo diferente. Podemos usar esse conhecimento em softwares que vão ser vendidos para uma gama muito diversificada de empresas, e essas possuem regras muito diferentes para cálculos de custos, insumos, impostos e comissões. Também é útil para o administrador do sistema, poder criar Plug-ins ou Workflows específicos para o seu caso.

Resumo do DevMan

Vamos explicar basicamente o que é metaprogramação e as diferentes abordagens que esse assunto pode ter. Citaremos alguns exemplos onde o leitor já usa metaprogramação e talvez não saiba. Vamos também criar ou simular situações onde seria necessário um interpretador de código, e criaremos oportunidades para o usuário do sistema programar.

Vitor Luiz Rubio
vitorrubio@gmail.com
http://www.vitorrubio.blogspot.com
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3.

Metaprogramação é uma palavra abrangente que pode ter diversos significados. Pode ser, por exemplo, a geração de código em uma linguagem qualquer. Se você tem um programa que gere código fonte para ser compilado por outro programa, isso é um caso de metaprogramação.

Um exemplo bastante conhecido por nós são os Templates e Wizards do

Delphi que nos ajudam a "digitar" certas porções de código. Um exemplo mais avançado seriam ferramentas como o Model Maker e o ECO ou qualquer outra ferramenta que gere código para nós.

Um outro exemplo de metaprogramação são os próprios compiladores, de qualquer linguagem, porque eles têm como objetivo gerar código de máquina a partir de um código fonte. O código binário resultante reflete as intenções do código fonte, portanto a criação de um compilador é um caso avançado de metaprogramação.

Alguns conceitos abordados aqui precisarão de explicação. Programa fonte é o nome que damos ao código fonte propriamente dito. Programa objeto seria o programa compilado. Linguagem de programação é a linguagem que usamos para criar o código fonte de um programa.

Quando estamos criando um compilador de uma linguagem qualquer o programa objeto é um compilador, ele passa a ser chamado de metaprograma, e a linguagem usada para criá-lo é chamada de metalinguagem. Chamamos de "linguagem objeto" a linguagem com a qual este compilador vai trabalhar.

Quando a linguagem objeto e a metalinguagem são a mesma, ou seja, uma linguagem pode ser sua própria metalinguagem, damos o nome de reflexão. Um ótimo exemplo é a nossa ferramenta preferida, o Delphi, que foi feito em Delphi, ou qualquer outro compilador de uma linguagem que foi criado nessa mesma linguagem.

A seguir explicaremos um pouco sobre duas outras formas em que a metaprogramação é usada. Nós nos concentraremos na segunda situação, mas observe que no nosso cotidiano usamos exemplos da primeira sem perceber.

Um programa com funções de compilador

Um compilador, ou uma biblioteca usada por ele, pode disponibilizar no programa compilado funções dele mesmo, para serem usadas no programa objeto.

Alguns compiladores disponibilizam uma API ou Interface para que programas compilados por ele possam usar funções do próprio compilador. Para ilustrar, imagine que você use a linguagem C++ para criar um outro compilador de C++. Posteriormente você transforma esse seu outro compilador de C++ de sua autoria em uma biblioteca, digamos uma DLL, para que possa ser incorporada e usada por qualquer outro programa feito por você.

Com isso você poderia criar programas num compilador de C++ normal, que usassem essa DLL para compilar outros programas de C++. Você poderia incorporar isso em um ERP para que administradores de sistemas criassem suas próprias rotinas.

No Delphi podemos ver esse exemplo no uso de genéricos. Genéricos usam funções do próprio compilador na metalinguagem Delphi. Além disso, RTTI e Genéricos são bons exemplos de reflexão, pois usam recursos do compilador para descobrir ou modificar informações do programa compilado.

Uma função que executa um comando

Sempre que uma biblioteca qualquer, em uma linguagem qualquer, possuir funções que aceitem como parâmetros expressões de texto ou Strings que contenham comandos, e interpretar ou executar esses comandos em tempo de execução, estamos lidando com um caso de metaprogramação.

Um exemplo bastante claro é um Browser. Consideremos o Mozilla Firefox. O Firefox foi escrito, em sua maior parte, em C/C++. Dentre as muitas funções de um Browser podemos destacar a habilidade de interpretar JavaScript.

JavaScript é uma linguagem de alto nível, muito poderosa e geralmente interpretada por um Browser. Programas podem ser escritos em JavaScript, principalmente para validação ou animação de formulários Web, mas não se limitando a isso.

Nesse caso especial do Browser podemos dizer que a metalinguagem foi C++ e que o metaprograma é o Browser. A linguagem alvo, ou mais precisamente linguagem objeto, é o JavaScript. É este o caso que vamos estudar.

Quem já programou em FoxPro, Visual Basic, Clipper ou JavaScript já conhece essa forma de metaprogramação. Essas linguagens têm funções ou recursos para executar comandos na própria linguagem. Esses comandos podem vir de arquivos de configuração, bancos de dados ou podem ser digitados pelo próprio usuário, e podem interagir com elementos já presentes no programa e fornecer uma ferramenta muito poderosa para alteração e customização dos programas. Um exemplo prático é a função eval do JavaScript, que pode ser vista na **Listagem 1**.

Listagem 1. Função Eval do Javascript

```
<script>
 eval("alert('Hello World');");
</script>
```

O JavaScript é uma linguagem que tem uma função especial, chamada eval que recebe como argumento uma String. Esta String pode ser uma fórmula matemática, assim eval retornaria o resultado, ou pode ser um código também em JavaScript.

O código da **Listagem 1** fará com que apareça a caixa de mensagem com o texto Hello World no Browser. Mas nós não usamos a função alert para fazer isso, nós usamos a função eval passando o comando alert em forma de String, como parâmetro. O que aconteceu foi que o interpretador do JavaScript, ao executar a função eval, usou recursos do próprio interpretador para executar também o conteúdo passado à eval. Com um input do tipo Text, ou uma TextArea, seria possível permitir ao usuário executar Scripts arbitrários para interagir com elementos da página. O que nós vamos fazer aqui é criar um programa que pode interpretar e executar comandos dados pelo usuário.

Cenários

Imagine que você tem um software para vender para empresas de ramos diferentes, com necessidades diferentes. Seria impossível atender a todas as demandas por alterações no

sistema. Caso tentássemos fazer isso nosso software ficaria com um número muito grande de configurações para fazer antes da implantação e um número muito grande de dependências e funcionalidades que seriam utilizadas por menos de 10% de nossos clientes.

Se nós mantemos apenas um único sistema, sem trabalhar com "específicos" é mais fácil garantir que nossos clientes estarão sempre com a versão mais atual do sistema, todos ao mesmo tempo, e uma correção ou melhoria no sistema pode ser atualizada e propagada rapidamente para todos os clientes.

De vez em quando surge um cliente com uma necessidade especial, a qual chamaremos de xxx. Se parecer uma boa ideia podemos nos sentir tentados a incorporar essa nova funcionalidade no sistema, mas sempre que fizermos isso temos de nos preocupar se essa funcionalidade não vai atrapalhar os demais clientes ou tornar o sistema invendável para clientes de outros ramos.

Como não queremos que isso aconteça, nós criamos no formulário de configuração do sistema mais um CheckBox, ou dois RadioButtons, com as opções "Ativar uso de xxx", "Não ativar uso de xxx". Uma segunda funcionalidade yyy dobraria o número de configurações para 4: "sim para xxx e para yyy", "não para xxx e yyy", "sim para xxx e não para yyy" e "não para xxx e sim para yyy". Uma terceira funcionalidade dobraria o número de possibilidades para 8 e assim criamos um verdadeiro inferno exponencial.

Isso pode se tornar muito perigoso, principalmente se você nunca usou boas práticas de POO, Interfaces e padrões de projeto. O seu sistema está preparado para trabalhar com tantos desvios diferentes de regras de negócio? As regras de negócio e de persistência dos dados estão isoladas em um mesmo lugar para que fique fácil replicar as configurações escolhidas ou modificadas? Ou você terá que varrer mais de 1000 formulários do sistema e procurar um trecho de código para colar um código como o da **Listagem 2**?

Listagem 2. Nunca faça isso: Copiar e colar um trecho de código em vários formulários

```
//substituir o código
if xxx then
//comandos
else
//comandos
//pelo código
if xxx and yyy then
//os dois ativados
else
if xxx then
//apenas xxx ativado
else
if yyy then
//apenas yyy ativado
else
//nenhuma das opções ativada
```

É incrível, mas isso acontece em muitos sistemas, de muitas empresas. E conforme o desenvolvedor vai modificando units em função de mudanças em outras, adicionando unidades em outras, ele vai criando dependências, acoplamentos e referências circulares que transformam o sistema num verdadeiro quebra-cabeça a ser resolvido. E se o usuário pudesse criar suas próprias rotinas para, digamos, automatizar tarefas?

Se o seu software tem uma funcionalidade que gera um relatório qualquer e o salva em um local escolhido pelo usuário, um destes usuários mais avançados talvez pudesse criar um Script para que esse relatório fosse posteriormente compactado e enviado por e-mail.

Muitos grandes ERP's são vendidos fechados, como uma caixa preta, com seu escopo mínimo, como se fosse um Framework, mas bem menos flexível que isso. Sendo assim, qualquer alteração de funcionalidade, a incorporação ou modificação de um relatório, ou qualquer outra necessária para atender os requisitos do cliente envolvem alterações, re-compilações, re-implementações. E se um bug for corrigido no "core" ou parte principal do sistema, podemos ter certeza que a correção ficará apenas em nossos próprios computadores e talvez no sistema do cliente que descobriu o bug. Os outros clientes, se não são afetados, nem ficam sabendo.

Por outro lado, existem bons ERP's no mercado que possuem consigo um interpretador de uma linguagem, para que clientes avançados ou consultores especializados possam, ao implantar o sistema em diferentes tipos de empresa, criar rotinas para customizar ou automatizar partes dos processos.

As ferramentas do Office são ótimos exemplos. Além dos recursos de macros, temos ao nosso dispor toda a estrutura da linguagem VBA e o que ela pode fazer. Lógico que sistemas inteiros não são criados nessas linguagens limitadas, mas pequenas rotinas de conversão ou integração de dados, por exemplo, o tratamento dos dados de um CRM antes de se fazer a carga em uma campanha de mail marketing.

Diferenças na forma de calcular o lucro líquido, calcular a comissão para vendedores e representantes, folha de pagamento e qualquer cálculo de imposto baseado em leis que podem mudar com o tempo ou que são diferentes de acordo com o estado e ramo de atividade da empresa.

Um exemplo clássico é o caso da padaria versus papelaria. Ambos têm seus impostos a calcular e pagar, mas na papelaria, enquanto canetas, pastas e mochilas são tributadas normalmente, os livros são isentos de alguns tipos de tributos. E o caso da padaria é até mais difícil. A farinha do pãozinho pode ser tributada, mas pode ter alguma espécie de incentivo, ou dissídio, do governo, então ela teria uma dedução no imposto.

Criando pontos de entrada para código do usuário

Vamos criar alguns pequenos programas que interpretem código em linguagem Pascal em certos pontos do sistema. Esse código pode vir de um arquivo ou do banco de dados, ou digitado na hora pelo cliente. Para isso usaremos um componente chamado Pascal Script for Delphi, da RemObjects. A RemObjects é uma empresa conhecida por fazer componentes, ferramentas e Frameworks avançados para Delphi, inclusive integrando-os a outros produtos. Eles criaram, por exemplo, o compilador do Delphi Prism. A linguagem Pascal ligeiramente diferente do Prism também é chamada de Oxygene.

Nota

Entre no site da RemObjects e clique na seção de Free Downloads. O endereço encontra-se disponível na seção de links no final do artigo. Baixe uma cópia do Pascal Script for Delphi e instale-a.

Após finalizar a instalação, teremos uma nova paleta adicionada ao nosso Delphi, como é possível ver na **Figura 1.** Agora crie uma nova aplicação Delphi para Win32 e salve o projeto como InterpretadorPascal, e ao Form principal vamos dar o nome de frmEditor e uFrmEditor.pas para a unit. Adicionaremos um Memo chamado mFonte e um botão chamado btRun, com o caption Executar.

Figura 1. Paleta Pascal Script

Vamos colocar no nosso formulário um componente TPSScript, o primeiro da paleta, e nomeá-lo como psExecutar. Um conceito importante a tratar aqui é o seguinte: o psExecutar não tem conhecimento de funções, procedures units e forms criados por você no seu programa. Estes precisam ser adicionados ao psExecutar. No começo ele á capaz apenas de lidar com Pascal puro, entretanto você pode escrever um programa inteiro e colocar para rodar. Mesmo que o programa funcione você não o verá rodando simplesmente porque não definiu as entradas e saídas do programa. No caso do Pascal, por exemplo, em modo console, temos como entrada o teclado ou Shell e como saída o console (tela de texto).

Outra coisa importante é que precisamos adicionar mais um Memo para armazenar os erros de compilação, senão não ficaremos sabendo se nosso script tem um erro. Coloque mais um Memo no formulário, chamado mMensagens e ainda outro chamado mSaida. O mMensagens nos dará as mensagens da execução e o mSaida será nosso console.

Para que possamos visualizar a saída no mSaida vamos escrever uma procedure estática global que aceitará como argumento uma string e a escreverá no memo mSaida. Vamos dar o nome a essa procedure de NossoWriteLn, como uma referência ao WriteLn do Pascal. Essa procedure será importada para dentro do nosso PascalScript e poderemos usá-la à vontade.

Nosso formulário ficará como na **Figura 2**.

Insira conforme a **Figura 2** dois botões, denominados como Executar e Testar. Este último apenas servirá para testar os métodos de integração com nosso Script antes de efetivamente usá-los. O código da **Listagem 3** será esta procedure que aceita uma String como entrada e a escreve no Memo de saída.

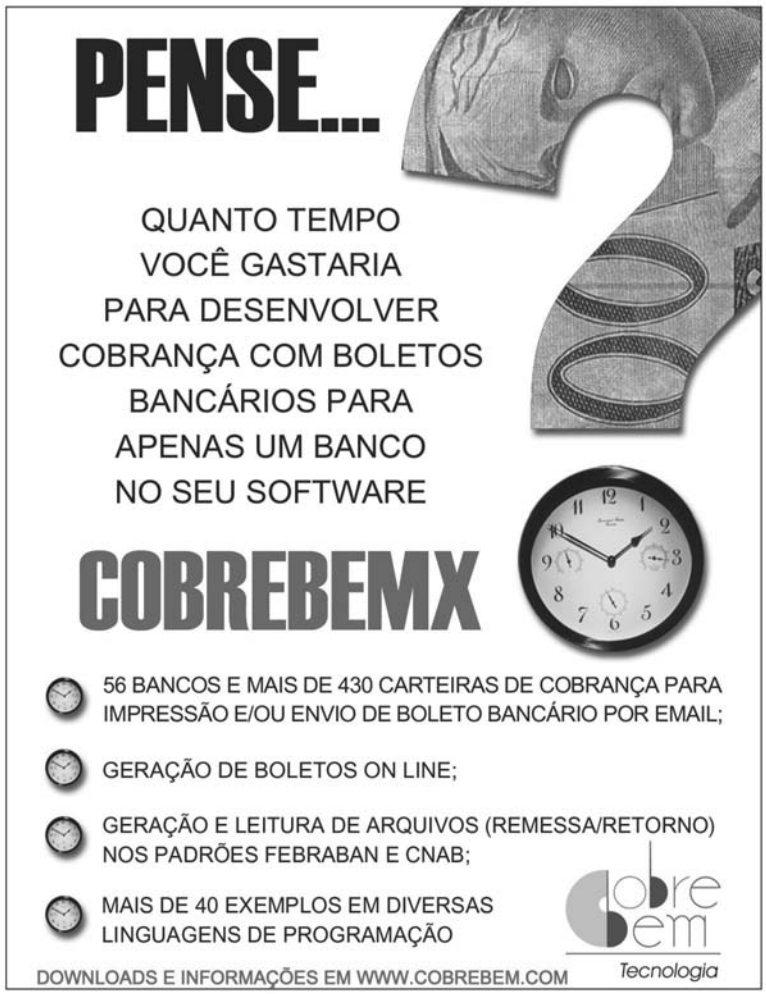
Esta procedure estática global faz parte da unit e não do form. Ela usa a variável pública global que contém a instância do nosso Form, o frmEditor, para acrescentar uma linha de texto em seu memo de saída. O código do botão Testar, na **Listagem 4**, mostra o que ela faz.

Nota do DevMan

Para usar a paleta de componentes no estilo do antigo Delphi 7, clique com o botão direito do mouse na barra de ferramentas e marque a opção Component. Esta barra de componentes possui um menu suspenso ao clicar com o botão direito do mouse que lhe permite escolher a paleta de forma vertical em ordem alfabética. Coisa que anteriormente só era possível com plug-ins de terceiros.

Nota

Nota: Para facilitar o uso do Script por usuários sem muito conhecimento de inglês ou programação, você pode dar o nome de "Escreva" a nossa procedure WriteIn, para que tenha uma notação semelhante à ensinada nos cursos de lógica de programação. Inclusive o nome da procedure dentro do nosso Script não necessita ser igual ao nome da procedure no executável, como veremos mais adiante.



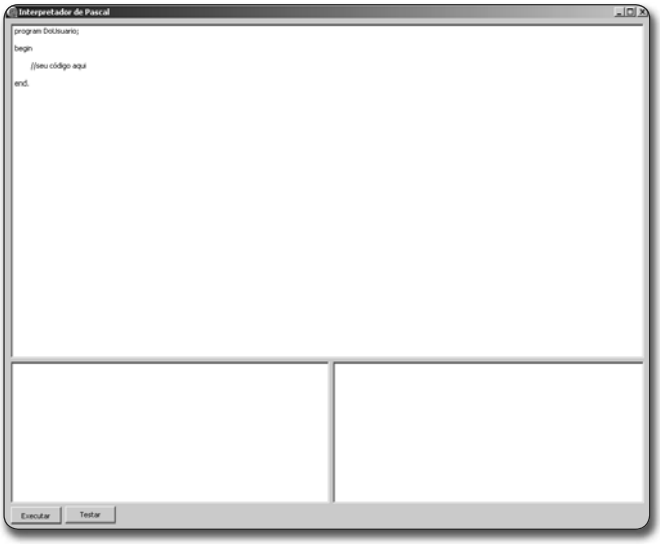


Figura 2. Formulário de edição de Script

Listagem 3. Procedure NossoWriteLn

```
procedure NossoWriteLn(const s: string);
begin
  frmEditor.mSaida.Lines.Add(s);
end;
```

Listagem 4. Botão btTestar conterá os testes de nossos métodos

```
procedure TfrmEditor.btTestarClick(Sender: TObject);
begin
  NossoWriteLn('Teste de saída');
end;
```

Este botão btTestar conterá os testes dos métodos do nosso programa que deverão ser usados dentro do Script. Uma vez testado, podemos importar o método. Para fazer isso edite o evento OnCompile do psExecutar e insira o código contido dentro da **Listagem 5** que ilustra como devemos fazer para incorporar um método do nosso programa dentro do Script.

Listagem 5. Importando métodos

```
procedure TfrmEditor.psExecutarCompile(Sender: TPSScript);
begin
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Writeln(s:
   string);');
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Escreva(s:
   string);');
end;
```

O método AddFunction do componente TPSScript aceita dois argumentos, um é um ponteiro para um método e o outro é uma String contendo o cabeçalho ou declaração do método. Para obter um ponteiro para um método, procedure ou function nós usamos o operador de endereço @. Isto significa que @NossoWriteLn é o endereço da função NossoWriteLn na memória de nosso programa. Você poderia aqui adicionar um ponteiro para qualquer tipo de função, contanto que o segundo argumento fosse um cabeçalho que refletisse o uso dela.

Com cabeçalho queremos dizer a declaração de uma função que usamos na seção interface de uma unit, sem a parte de sua implementação. Por exemplo:

```
procedure NossoWriteLn(const s: string);
```

Incorporamos no nosso interpretador duas vezes a mesma função, uma com nome de WriteLn e outra com o nome de Escreva, para ilustrar que o nome não importa, contanto que a assinatura seja a mesma.

Para podermos digitar um Script e visualizar sua execução precisamos antes definir qual será o Script a ser interpretado, compilar, verificar por mensagens de erro e finalmente executar. A **Listagem 6** contém o código do botão executar, que mostrará os passos que devemos dar.

Primeiro de tudo nós limpamos os Memos de mensagens e saídas. Depois atribuímos à propriedade Script do psExecutar o código fonte que digitamos no mFonte. Criamos duas variáveis booleanas, uma para armazenar o resultado da compilação, que receberá o valor verdadeiro para sucesso e falso para erro, e a outra para armazenar, analogamente, o resultado da execução. A variável i, do tipo inteiro, serve para percorrer a lista de mensagens de compilação, sejam erros ou não, varrendo-as com um laço contado e mostrando-as no Memo mMensagens.

Caso as duas variáveis booleanas forem verdadeiras adicionamos uma mensagem de sucesso, caso contrário usamos os métodos ExecErrorToString para obter a mensagem de erro em tempo de execução. Utilizamos ExecErrorProcNo para saber em que método foi o erro e ExecErrorByteCodePosition para saber em qual linha.

Para testar nosso interpretador de Pascal digite o código da **Listagem 7** no mFonte, com o programa em execução, e clique no botão executar.

Listagem 6. Compilando e executando

```
procedure TfrmEditor.btExecutarClick(Sender: TObject);
var
  Compilou, Executou: boolean;
  i: integer;
begin
  mMensagens.Clear;
  mSaida.Clear;
  psExecutar.Script.Text := mFonte.Text;
  Compilou := psExecutar.Compile;
  if Compilou then
  begin
    mMensagens.Lines.Add('Programa compilado com sucesso!');
    Executou := psExecutar.Execute;
    if Executou then
    begin
      mMensagens.Lines.Add('Programa executado com sucesso!');
    end
    else
    begin
      mMensagens.Lines.Add('Ocorreu o erro de execução: ' +
        psExecutar.ExecErrorToString +' onde? '+
        Inttostr(psExecutar.ExecErrorProcNo)+'.'+
        Inttostr(psExecutar.ExecErrorByteCodePosition));
    end;
  end
  else
     mMensagens.Lines.Add('Erro de compilação:');

  for i := 0 to psExecutar.CompilerMessageCount - 1 do
  begin
    mMensagens.Lines.Add('Compilador: '+ psExecutar.
     CompilerErrorToStr(i));
  end;
end;
```

Listagem 7. Programa do usuário (digite no Memo)

```
program DoUsuario;

begin
   //seu código aqui
   escreva('alo mundo');
   writeln('hello world');
end.
```

Aqui testamos os dois nomes que demos a nossa procedure WriteLn. Isso produzirá como saída do compilador as mensagens:

```
Programa compilado com sucesso!
Programa executado com sucesso!
```

E como saída do programa:

```
alo mundo
hello world
```

Agora vamos inserir um erro proposital para ver o que acontece. Vamos declarar duas variáveis do tipo inteiro, somá-las e tentar mostrar o resultado. Digite o código da **Listagem 8** novamente no Memo.

Listagem 8. Soma (digite no memo)

```
program DoUsuario;
var
  x, y, r: integer;
begin
    //seu código aqui
    x := 5;
    y := 9;
    r := x+y;
    escreva(r);
end.
```

O programa da **Listagem 8** gera o seguinte erro de compilação:

```
Erro de compilação:
Compilador: [Error] (14:18): Type mismatch
```

Isso era esperado, uma vez que passamos para a função escreva a variável r, que é inteiro, sendo que a função "escreva" só funciona com Strings. O que devemos fazer então? Usar IntToStr? Vamos tentar, conforme a **Listagem 9**.

Listagem 9. Soma com IntToStr (digite no memo)

```
program DoUsuario;
var
  x, y, r: integer;
begin
    //seu código aqui
    x := 5;
    y := 9;
    r := x+y;
    escreva(inttostr(r));
end.
```

Como vimos o código da **Listagem 9** produziu a saída esperada, com o resultado de 14. Estamos já percebendo o poder dessa ferramenta. Vamos agora fazer um teste mais real, permitindo que o usuário entre com valores. Para isso, vamos criar a nossa procedure ReadLn, conforme a **Listagem 10**.

Listagem 10. Implementação de ReadLn

```
procedure NossoReadLn(var s: string);
begin
  s := InputBox('Digite um valor:', 'Digite um valor:', '');
end;
```

Temos que adicionar essa procedure para que o nosso código a encontre e a reconheça, da mesma forma como adicionamos a outra. Na **Listagem 11** podemos ver o código completo até aqui.

Listagem 11. Adicionando o ReadLn

```
procedure TfrmEditor.psExecutarCompile(Sender: TPSScript);
begin
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Writeln(s:
   string);');
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Escreva(s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure ReadLn(var s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure Leia(var s:
   string);');
end;
```

O código da **Listagem 12** insere duas variáveis do tipo string para receber os valores digitados pelo usuário e as converte para inteiro para realizar os cálculos.

Listagem 12. Soma permitindo que o usuário entre com os valores (digite no memo)

```
program DoUsuario;
var
  x, y, r: integer;
  sx, sy: string;
begin
    //seu código aqui
    leia(sx);
    leia(sy)
    x := strtoint(sx);
    y := strtoint(sy);
    r := x+y;
    escreva(inttostr(r));
end.
```

Vamos agora começar a interagir mais com os outros recursos do Delphi. Vamos ver se conseguimos executar a função ShowMessage() com a mensagem Alo mundo. Digite o código da **Listagem 13** no Memo e observe o resultado.

Listagem 13. Usando ShowMessage (digite no memo)

```
program DoUsuario;
begin
  ShowMessage('Alo Mundo');
end.
```

Como esperado, o resultado diz que a função ShowMessage não foi encontrada.

```
Erro de compilação:
Compilador: [Error] (5:1): Unknown identifier 'ShowMessage'.
```

Basta adicionarmos a função ShowMessage, conforme adicionamos as outras, não sendo necessária sua criação, pois esta já existe. Coloque como a seguir no evento psExecutarCompile:

```
psExecutar.AddFunction(@ShowMessage, 'procedure ShowMessage(s: string);');
```

Adicionando Application ao programa interpretado

Passando a exemplos mais avançados, vamos interagir com o nosso próprio Form. Para fazer isso, no uses da seção implementation coloque todas essas units que vêm junto com o componente, conforme a **Listagem 14**.

Listagem 14. Units que adicionam units dentro do programa interpretado

```
implementation
uses
  uPSR_std,
  uPSC_std,
  uPSR_stdctrls,
  uPSC_stdctrls,
  uPSR_forms,
  uPSC_forms,
  uPSC_graphics,
  uPSC_controls,
  uPSC_classes,
  uPSR_graphics,
  uPSR_controls,
  uPSR_classes;
```

Estas units servem para registrar para o uso do programa interpretado as units padrão do Delphi que usamos em quase todas as aplicações, sendo elas: System, SysUtils, Forms, Graphics, Classes e Controls. Geralmente quando criamos uma nova aplicação em Delphi a própria IDE adiciona automaticamente essas units mais usadas para nós. No nosso interpretador temos que adicioná-las manualmente. Essas units que colocamos na seção uses vão registrar, internamente, as units mais usadas dentro do escopo de nosso programa interpretado. As units começadas com uPSC são as que fazem o registro a ser usado em tempo de compilação e as uPSR fazem o registro das units usadas em Runtime. Além destas verifique também se as units uPSCompiler e uPSRuntime estão no uses da seção Interface.

O componente TPSScript possui dois eventos, que acontecem logo antes de compilar e logo antes de executar. Usaremos esses eventos para adicionar algumas informações no nosso TPSScript, usando métodos dessas units que acabamos de inserir. Edite o evento onCompImport e adicione o código da **Listagem 15**.

Listagem 15. Adicionando informações de compilação

```
procedure TfrmEditor.psExecutarCompImport(Sender: TObject;
  x: TPSPascalCompiler);
begin
  SIRegister_Std(x);
  SIRegister_Forms(x);
end;
```

O método SIRegister_Std está na unit uPSR_Std e serve para registrar no compilador (através do argumento x passado pelo método do evento) as units padrão do Delphi. O método SIRegister_Forms registra a unit Forms.

Se olharmos os fontes da VCL (arquivos .pas que vem junto com o Delphi) veremos que a unit Forms, além da classe TForm, tem várias outras classes e tipos de apoio, por exemplo a classe TApplication e a variável pública global Application.

Analogamente, na **Listagem 16**, adicionaremos informações que nosso interpretador precisará em Runtime. Vamos editar o evento onExecImport e adicionar o código da **Listagem 16**.

Este código faz exatamente a mesma coisa que a **Listagem 15**, mas neste caso é para ser usado em tempo de execução. Nos nossos exemplos os dois serão necessários.

Agora teremos que dizer para o compilador que existirá uma variável chamada Application, do tipo TApplication. Para isso, modificaremos o evento onCompile, conforme a **Listagem 17**.

Listagem 16. Adicionando informações de execução

```
procedure TfrmEditor.psExecutarExecImport(Sender: TObject; se:
 TPSExec;
  x: TPSRuntimeClassImporter);
begin
  RIRegister_Std(x);
  RIRegister_Forms(x);
end;
```

Listagem 17. Registrando variáveis globais

```
procedure TfrmEditor.psExecutarCompile(Sender: TPSScript);
begin
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Writeln(s:
   string);');
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Escreva(s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure ReadLn(var s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure Leia(var s:
   string);');
  psExecutar.AddFunction(@ShowMessage, 'procedure ShowMessage(s:
   string);');
  //adicionamos isso para existir a variável
  psExecutar.AddRegisteredVariable('Application', 'TApplication');
end;
```

O método AddRegisteredVariable adiciona uma variável e um tipo. Porém se executarmos o programa agora, Application será nil, não será a mesma Application do nosso programa.

Existe um método capaz de atribuir valores de objetos em tempo de execução. É o método SetVarToInstance, que devemos usar no evento OnExecute do psExecutar. Então vamos editar o evento onExecute e modificá-lo conforme a **Listagem 18**.

Listagem 18. Passando uma variável do sistema para o script

```
procedure TfrmEditor.psExecutarExecute(Sender: TPSScript);
begin
  psExecutar.SetVarToInstance('Application', Application);
end;
```

O método SetVarToInstance aceita dois argumentos, uma String que será o nome de uma variável e uma instância. Ao iniciar a execução do programa a instância padrão de TApplication armazenada em Application do nosso sistema será passada para a variável Application do Script. Com isso podemos exibir o título ou caminho de nossa aplicação e até manipular Forms, conforme veremos na **Listagem 19**.

Listagem 19. Programa interagindo com Application

```
program DoUsuario;
begin
    escreva(Application.Exename);
    escreva(Application.Title);
    Application.MainForm.Caption := 'Customização do programa';
end.
```

Adicionando Self ao programa interpretado

Para adicionarmos Self ao nosso programa, onde Self se referirá a algum Form, por exemplo, o próprio formulário do editor, devemos no onCompile registrar a variável Self do tipo TForm:

```
psExecutar.AddRegisteredVariable('Self', 'TForm');
```

Nota do DevMan

A versão corrente do Pascal Script trabalha com AnsiStrings e AnsiChars, comuns até o Delphi 2007, onde cada Char tinha um Byte. Porém toda String no Delphi 2009 e 2010 é por padrão WideString, de dois Bytes cada WideChar. Então, para garantir a compatibilidade e usar algumas funções corretamente, crie uma função no executável que seja réplica da função que você deseja usar, porém com Ansi String. Você até pode internamente converter AnsiString em WideString e chamar a função original, mas deve registrar a sua função modificada.

E depois atribuir a ela o valor da instância do nosso Form, no evento OnExecute:

```
psExecutar.SetVarToInstance('Self', Self);
```

Da mesma forma como já fizemos para Application. Isso torna possível o código da **Listagem 20**.

Listagem 20. Mudando o caption do Form (digite no memo)

```
program DoUsuario;
begin
   self.caption := 'Em execução!';
end.
```

Vamos adicionar um suporte bem mais completo às ferramentas do Delphi. Adicione o código das **Listagens 21** e **22** aos eventos OnCompImport e OnExecImport respectivamente. Depois de adicionar essas units, propriedades como Color passarão a funcionar.

Listagem 21. Importação das outras units em tempo de compilação

```
procedure TfrmEditor.psExecutarCompImport(Sender: TObject;
  x: TPSPascalCompiler);
begin
  SIRegister_Std(x);
  SIRegister_Classes(x, true);
  SIRegister_Graphics(x, true);
  SIRegister_Controls(x);
  SIRegister_stdctrls(x);
  SIRegister_Forms(x);
end;
```

Listagem 22. Importação das outras units em tempo de execução

```
procedure TfrmEditor.psExecutarExecImport(Sender: TObject; se:
TPSExec;
  x: TPSRuntimeClassImporter);
begin
  RIRegister_Std(x);
  RIRegister_Classes(x, True);
  RIRegister_Graphics(x, True);
  RIRegister_Controls(x);
  RIRegister_stdctrls(x);
  RIRegister_Forms(x);
end;
```

DataPump

Sempre que implantamos um sistema novo em um cliente que já possuía um sistema, há a necessidade de fazer a migração dos dados. Muitas vezes este processo é até simples. Se estiverem no mesmo banco de dados que vamos usar, fazemos algumas Queries, alguns Inserts e algumas Stored Procedures.

Nota do DevMan

Embora não esteja relacionado diretamente com o assunto do artigo, é interessante considerar o uso do componente TSynEdit. O TSynEdit é um componente gratuito, de código fonte livre sob licença GPL que parece um TMemo especial, mas ele faz Syntax Highlighting (destaque de sintaxe) de muitas linguagens, totalmente configurável, com uma régua à esquerda que exibe o número da linha, quebra automática de linha passando de um determinado tamanho e muitos outros recursos avançados. Ele pode ser usado para a edição do código-fonte do programa a ser interpretado e ele deve ser usado caso deseje fazer o Debug do Script. Os exemplos que vêm junto com o pacote de componentes Pascal Script usam o SynEdit junto com o TPsScriptDebugger para fazer a depuração do Script.

Em alguns casos precisamos usar ferramentas de DTS (Data Transformation Services) do MS SQL Server, montar Scripts de inserção em fórmulas do Excel e até usar o Access para se conectar em duas bases de dados e passar os dados de uma para outra, isso quando o próprio Access não é uma das bases.

Há também os casos em que o banco a ser migrado é DBASE ou Paradox. Se nosso trabalho simplesmente se limitasse à cópia dos dados seria fácil, mas muitas vezes temos de aplicar fórmulas, juntar dados de tabelas diferentes ou distribuir dados em tabelas diferentes.

Um cenário comum é encontrarmos tabelas onde:

- Pessoa física e jurídica residem na mesma tabela;
- Os documentos CPF e CNPJ não são validados e são salvos com as máscaras;
- Caracteres M para sexo masculino, F para feminino e J para pessoa jurídica;
- Uma Flag que pode ser F para física e J para jurídica;
- Contatos na mesma tabela;
- Endereços na mesma tabela;

Supondo agora que vamos migrar e organizar esses dados, podemos fazer uma única aplicação genérica que abra ou se conecte em qualquer banco de dados, colocando nessa aplicação o suporte a Scripts personalizados para que possamos levar o mesmo programa a todos os clientes e customizar a migração dos dados com um Script, conforme necessário.

Como último exemplo vamos criar um banco de dados para simular o sistema antigo de nossos clientes e outro para simular nosso sistema. Não usaremos nem MS SQL Server e nem Firebird para ganhar tempo, pois o banco de dados em si não é o foco do nosso artigo. Criaremos dois bancos de dados em Access, um chamado origem (antigo) e outro chamado destino (novo). Assim será mais fácil para levar os arquivos do banco de um lugar para o outro. Para isso utilizaremos o Office 2003. No banco de dados de origem crie uma tabela chamada clientes, conforme a **Figura 3**.

O correto neste caso seria dividir os endereços entre tipo de logradouro, logradouro, cep, bairro, cidade, uf, complemento e assim por diante, mas nosso exemplo é apenas ilustrativo, para fins didáticos. Essa tabela será dividida em três outras tabelas como destino: PessoaJuridica, PessoaFisica e Enderecos.

clientes

Nome do campo	Tipo de dados
ID	Número
Nome	Texto
Documento	Texto
Endereco1	Texto
Endereco2	Texto
Fone1	Texto
Fone2	Texto
Sexo	Texto

Figura 3. Tabela de Clientes

Nota

Nem todo mundo utiliza separar pessoas físicas e jurídicas em duas tabelas. O que não é o caso do endereço, na qual é bem comum possuir uma decomposição em diversas tabelas. Alguns sistemas têm endereço de entrega, cobrança e faturamento na mesma tabela, permitindo três endereços por pessoa. Outros sistemas, como e-commerces permitem múltiplos endereços por pessoa, pois a pessoa pode sempre mandar entregar em um local diferente.

Insira os dados na tabela, conforme a **Figura 4**.

clientes

ID	Nome	Documento	Endereco1	Endereco2	Fone1	Fone2	Sexo	Adicionar Novo Campo
1	Cliente 1	155.803.240-14	rua a1	rua b1	00000000	11111111	F	
2	Cliente 2	342.638.980-01	rua a2	rua b2	11111111	22222222	f	
3	Cliente 3	238.545.786-52	rua a3	rua b3	33333333	44444444	M	
4	Cliente 4	834.711.317-36	rua a4	rua b4	55555555	66666666	m	
5	Cliente 5	523.538.462-89	rua a5	rua b5	77777777	88888888	m	
6	cliente 6	375.018.967-63	rua a6	rua b6	99999999	10101010	M	
7	cliente 7	308.641.087-61	rua a7	rua b7	12121212	13131313	F	
8	cliente 8	063.571.112-52	rua a8	rua b8	14141414	15151515	F	
9	cliente 9	565.158.611-33	rua a9	rua b9	16161616	17171717	F	
10	cliente 10	502.471.934-45	rua a10	rua b10	18181818	19191919	f	
11	empresa 1	48.421.842/00(	praça c1	alameda d1	20202020	21212121	J	
12	empresa 2	60.520.580/00(	praça c2	alameda d2	23232323	24242424	J	
13	empresa 3	68.993.462/00(	praça c3	alameda d3	25252525	26262626	J	
14	empresa 4	58.816.204/00(	praça c4	alameda d4	27272727	28282828	J	
15	empresa 5	76.692.224/00(	praça c5	alameda d5	29292929	30303030	J	
16	empresa 6	38.624.142/00(	praça c6	alameda d6	31313131	32323232	J	
17	empresa 7	46.635.243/00(	praça c7	alameda d7	34343434	35353535	J	
18	empresa 8	92.840.933/00(	praça c8	alameda d8	36363636	37373737	J	
19	empresa 9	65.684.363/00(	praça c9	alameda d9	38383838	39393939	J	
20	empresa 10	62.833.603/00(	praça c10	alameda d10	40404040	40414141	J	
*	0							

Figura 4. 20 registros de clientes

Estes CPF's e CNPJ's são fictícios e aleatórios, gerados praticamente com o mesmo algoritmo de se validar CPF ou CNPJ. O banco de dados destino terá as tabelas endereços, conforme a **Figura 5**, PessoaFisica conforme a **Figura 6** e PessoaJuridica conforme a **Figura 7**.

Trata-se apenas da separação da origem em várias outras tabelas. A ideia em relação a tabela de endereços é que se o número de endereços não for fixo, puder ser nulo, e puderem existir 0 ou n, não vale a pena colocar esses campos nas tabelas de pessoa física e pessoa jurídica.

Para fazer a relação do endereço com a devida PessoaFisica ou PessoaJuridica podemos usar uma chave estrangeira, se o gerador dos códigos das duas tabelas seguirem a mesma sequência, ou podemos usar duas chaves, sendo uma para cada tabela.

Não usamos campos do tipo autonumeração aqui simplesmente para facilitar a cópia dos dados em outra tabela mantendo-se as relações.

Criadas as tabelas, vamos nos conectar a elas com os componentes da paleta dbGo (ADO). Coloque no seu formulário quatro componentes TAdoDataSet, dando-lhes os nomes de AdoOrigem e AdoDestinoPF, AdoDestinoPJ e AdoDestinoEnderecos, respectivamente. A propriedade connection string do AdoOrigem deverá ser:

Nome do campo	Tipo de dados
ID	Numeração Automátic
CodCli	Texto
Endereco	Texto
Telefone	Texto

Figura 5. Tabela de endereços no destino

Nome do campo	Tipo de dados
ID	Número
CPF	Texto
Nome	Texto
Sexo	Número

Figura 6. Tabela de pessoas físicas no destino

Nome do campo	Tipo de dados
ID	Número
CNPJ	Texto
Nome	Texto

Figura 7. Tabela de pessoas jurídicas no destino

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=<<Caminho>>\origem.
mdb;Persist Security Info=False
```

E para os outros três:

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=<<Caminho>>\destino.
mdb;Persist Security Info=False
```

O seu formulário ficará como na **Figura 8**.

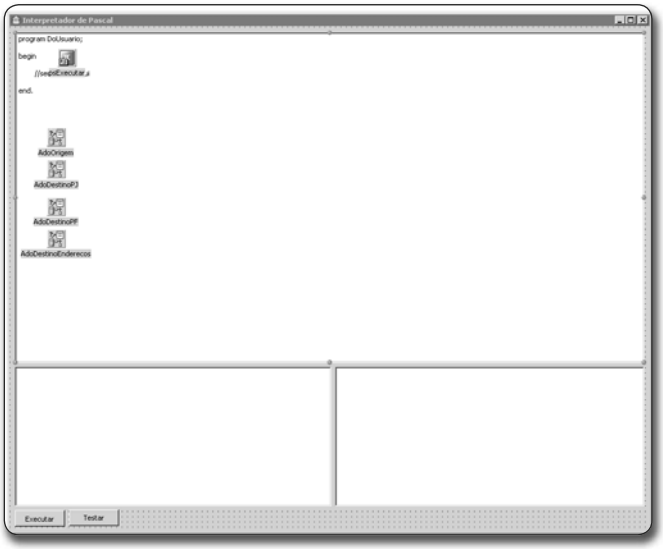


Figura 8. Formulário com os componentes de acesso ao Access

Agora vamos editar as propriedades CommandText dos quatro TAdoDatasets e adicionar os campos TFields.

O AdoOrigem terá a SQL:

```
select * from clientes
```

O AdoDestinoPJ terá a SQL:

```
select * from PessoaJuridica
```

O AdoDestinoPF terá a SQL:

```
select * from PessoaFisica
```

E por último o AdoDestinoEnderecos terá a SQL:

```
select * from Enderecos
```

Feito isso, vamos alterar novamente os eventos OnCompImport e OnExecImport do psExecutar para adicionarmos os quatro AdoDataSets, seus tipos e variáveis. Estamos registrando os AdoDataSets como simples TDataSets, explicaremos o porquê disso mais adiante. O código dos eventos ficará como na **Listagem 23**.

Listagem 23. Registrando os AdoDataSets

```
procedure TfrmEditor.psExecutarCompile(Sender: TPSScript);
begin
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Writeln(s:
   string);');
  psExecutar.AddFunction(@NossoWriteLn, 'procedure Escreva(s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure ReadLn(var s:
   string);');
  psExecutar.AddFunction(@NossoReadLn, 'procedure Leia(var s:
   string);');
  psExecutar.AddFunction(@ShowMessage, 'procedure ShowMessage(s:
   string);');

  //adicionamos isso para existir a variável
  psExecutar.AddRegisteredVariable('Application', 'TApplication');
  psExecutar.AddRegisteredVariable('Self', 'TForm');

  psExecutar.AddRegisteredVariable('AdoOrigem', 'TDataSet');
  psExecutar.AddRegisteredVariable('AdoDestinoPJ', 'TDataSet');
  psExecutar.AddRegisteredVariable('AdoDestinoPF', 'TDataSet');
  psExecutar.AddRegisteredVariable('AdoDestinoEnderecos', 'TDataSet');
end;

procedure TfrmEditor.psExecutarExecute(Sender: TPSScript);
begin
  psExecutar.SetVarToInstance('Application', Application);
  psExecutar.SetVarToInstance('Self', Self);

  psExecutar.SetVarToInstance('AdoOrigem', AdoOrigem);
  psExecutar.SetVarToInstance('AdoDestinoPJ', AdoDestinoPJ);
  psExecutar.SetVarToInstance('AdoDestinoPF', AdoDestinoPF);
  psExecutar.SetVarToInstance('AdoDestinoEnderecos',
  AdoDestinoEnderecos);
end;
```

Adicione no formulário um componente TPsImport_DB (PSImport_DB1) da paleta Pascal Script conforme a **Figura 9** e adicione ele à lista de Plugins do psExecutar. Para isso selecione o psExecutar, abra o Object Inspector e clique na propriedade Plugins do psExecutar, a **Figura 10** ilustra isso. Clique no botão Add e será criado um novo Plugin. Selecionando o Plugin aparecerá no Object Inspector a lista de Plugins que se encontram no formulário. As **Figuras 11** e **12** concluem o processo.

Isso é necessário para que o pascal Script tome conhecimento da existência da unit DB e os tipos criados dentro dela. Assim ele registrará a unit DB e poderemos usar tipos como TDataSet, acessar Fields e propriedades como FieldByName por exemplo.

Repita os mesmos passos das **Figuras 9** a **12** para adicionar um segundo Plugin, o PSImport_Classes(PSImport_Classes1). Este Plugin é necessário para registrar Classes e tipos da unit Classes. Ele insere para uso do psExecutar outras classes e tipos da unit classes que os métodos SIRegister_Classes e RIRegister_Classes não registram. Isso é necessário para que certos tipos de dados possam ser usados em Runtime pelo psExecutar.

Figura 9. O Componente TPsImport_DB

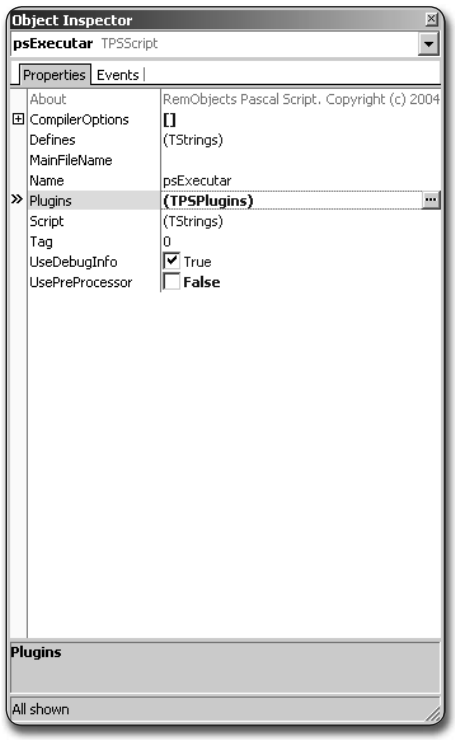


Figura 10. Propriedade TPSPlugins

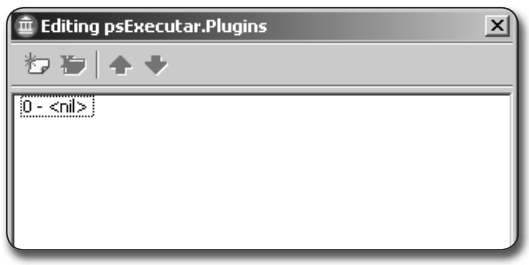


Figura 11. O editor de plugins

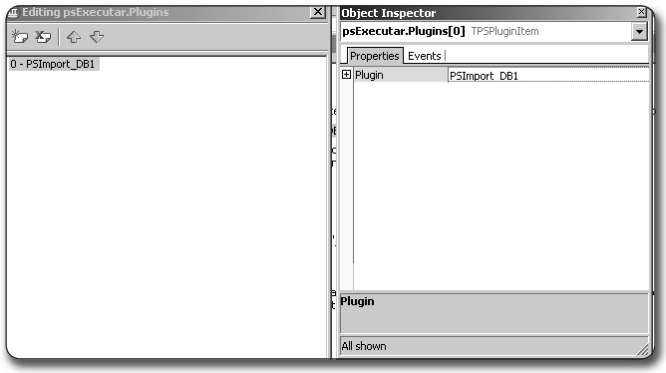


Figura 12. O plugin adicionado

Dica

Na paleta Pascal Script existem muitos outros componentes interessantes que valem a pena estudar, como Plugins para usar funções de dll's e integração com COM.

Com os dois componentes o formulário ficará como na **Figura 13**.

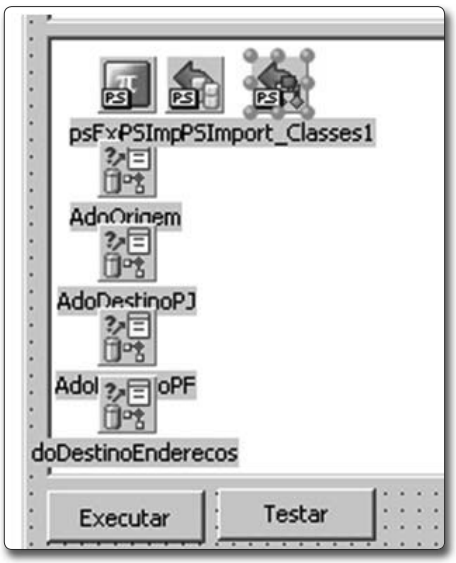


Figura 13. Com os plugins e AdoDatasets

É importante manter uma determinada ordem nos Plugins, por exemplo, o Plugin de classes deve vir antes que o de DB, já que no próprio Delphi a unit DB depende da unit Classes. Para isso mova os Plugins conforme a **Figura 14.**

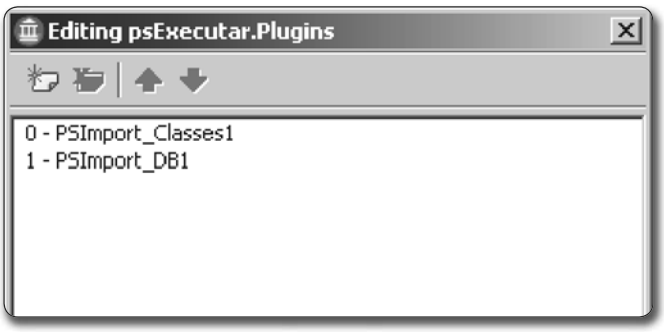


Figura 14. Ordem correta dos plugins

Digite no mFonte o código da **Listagem 24** e abra o banco de destino no Access, veja que os dados foram validados e organizados nas respectivas tabelas.

O que fizemos aqui foi construir um programa para migração/conversão de dados que pode ser customizado sem a necessidade de re-compilar. Primeiro abrimos os DataSets envolvidos. Repare que registramos os AdoDataSets como TDataSets por dois motivos, o AdoDataset é na verdade uma descendente de TDataSet e também porque o Pascal Script conhece o básico das bibliotecas, classes e componentes do Delphi. Ele sabe o que é um DataSet, mas não o que é um AdoDataSet.

Listagem 24. Script para trabalhar com BD (digite no memo)

```
program DoUsuario;
var
//armazena temporariamente o documento, para limpar os
caracteres e analisar o tipo de pessoa
documento: string;
//retira os caracteres de pontuação do cpf ou cnpj
function ApenasNumerosStr(pStr: string): string;
var
  I: Integer;
begin
  Result := '';
  for I := 1 to Length(pStr) do
    if pStr[I] in ['1', '2', '3', '4', '5', '6', '7', '8', '9',
     '0'] then
      Result := Result + pStr[I];
end;
//retorna 1 para física e 2 para jurídica, verificaremos apenas
baseado na quantidade de caracteres, não faremos validação.
function FisJur(documento: string): integer;
begin
    case Length(documento) of
        11: Result := 1;
        14: Result := 2;
    end;
end;
begin
//abre os destinos
AdoDestinoPj.Open;
AdoDestinoPf.Open;
AdoDestinoEnderecos.open;
//abre a origem e posiciona o cursor
AdoOrigem.open;
AdoOrigem.first;
while not AdoOrigem.eof do
 try
     try
        //adicionando 2 endereços para cada cliente
        AdoDestinoEnderecos.insert;
        AdoDestinoEnderecos.fieldByName('CodCli').AsInteger :=
         AdoOrigem.fieldByName('ID').AsInteger;
        AdoDestinoEnderecos.fieldByName('Endereco'). AsString:=
         AdoOrigem.fieldByName('Endereco1'). AsString;
        AdoDestinoEnderecos.fieldByName('Telefone'). AsString:=
         AdoOrigem.fieldByName('fone1'). AsString;
        AdoDestinoEnderecos.post;
        AdoDestinoEnderecos.insert;
        AdoDestinoEnderecos.fieldByName('CodCli').AsInteger :=
         AdoOrigem.fieldByName('ID').AsInteger;
        AdoDestinoEnderecos.fieldByName('Endereco'). AsString:=
         AdoOrigem.fieldByName('Endereco2'). AsString;
        AdoDestinoEnderecos.fieldByName('Telefone'). AsString:=
         AdoOrigem.fieldByName('fone2'). AsString;
        AdoDestinoEnderecos.post;

        documento :=  ApenasNumerosStr( AdoOrigem.fieldByName
        ('documento').asstring);

        if FisJur( documento   ) = 1 then
        begin
            AdoDestinoPF.insert;
            AdoDestinoPF.fieldByName('ID').Asinteger :=
             AdoOrigem.fieldByName('ID').AsInteger;
            AdoDestinoPF.fieldByName('CPF').AsString := documento;
            AdoDestinoPF.fieldByName('Nome').AsString :=
             AdoOrigem.fieldByName('Nome').AsString;
            if uppercase(AdoOrigem.fieldByName('Sexo').AsString) =
            'M' then
                AdoDestinoPF.fieldByName('Sexo').Asinteger :=  1
            else
            AdoDestinoPF.fieldByName('Sexo').Asinteger :=  2;
            AdoDestinoPF.post;
         end
        else
        begin
            AdoDestinoPj.insert;
            AdoDestinoPj.fieldByName('ID').Asinteger :=
             AdoOrigem.fieldByName('ID').AsInteger;
            AdoDestinoPj.fieldByName('CNPJ').AsString := documento;
            AdoDestinoPj.fieldByName('Nome').AsString :=
             AdoOrigem.fieldByName('Nome').AsString;
            AdoDestinoPj.post;
        end;

    except

    end;
 finally
    adoorigem.next;
 end;

//fechando todas as tabelas
AdoDestinoPj.Close;
AdoDestinoPf.Close;
AdoDestinoEnderecos.Close;
AdoOrigem.Close;
end.
```

Depois de abertos, nós varremos o DataSet de origem com um laço while. Colocamos o next dentro do bloco finally para garantir que o programa tentará gravar o próximo registro mesmo se o registro corrente causar algum erro. Assim não paramos o loop a cada exception.

A cada iteração do laço nós criamos um registro na tabela apropriada com insert e salvamos com post. Dentro do laço é capturado o código do cliente e gravado na tabela de endereços dois endereços para cada cliente. A tabela cliente de origem tinha endereco1 e endereco2 desnormalizados, no mesmo nível. Já a tabela de endereços do destino tem uma relação de 1 para N com a tabela de clientes. Cada cliente pode ter N endereços não se limitando a 2. Por isso a cada iteração do laço pela tabela de origem nós obtemos e gravamos os dois endereços de uma vez na tabela de destino de endereços, gravando também o código do cliente.

Durante a varredura nós limpamos o documento, para conter apenas números, sem máscaras. Verificamos através da função Length o tamanho do documento, se ele é um CPF, que possui 11 dígitos ou um CNPJ, que possui 14 dígitos. Se for CPF é uma pessoa física, se for CNPJ é uma pessoa jurídica. Gravamos cada tipo de cliente em sua tabela apropriada.

Para clientes pessoa física verificamos o sexo, M ou F, e gravamos como 0 ou 1, apenas para ilustrar o que pode ser feito nesses casos onde o significado do dado é o mesmo mas o tipo de dado armazenado é diferente, e um simples DataPump não resolveria mesmo se o nome das colunas fosse o mesmo.

Depois de executar o programa digitado no mFonte você pode abrir os bancos de dados e comparar. Veja que o destino está populado com os dados em forma e estrutura corretas. Ficaremos com 40 endereços e 10 clientes de cada tipo.

Dica

Você também pode usar Scripts personalizados para tratar ou validar as entradas do usuário, de maneira personalizada, sinalizando os erros com mensagens e cores.

Conclusão

Metaprogramação é um assunto estudado pela ciência da computação, tão antigo quanto a própria programação em si. Neste artigo mal chegamos a arranhar a superfície do assunto. Como vimos, além da interpretação de código em Runtime, também existem recursos do compilador disponíveis ao programador e ao programa, como tipos genéricos. Alguns autores afirmam que as macros do C/C++ e diretivas de compilação (ou conditional defines) também podem ser enquadrados em metaprogramação, mesmo que em outro foco.

É natural que linguagens interpretadas, como JavaScript, Python, Ruby e outras tenham muito mais recursos de metaprogramação do que linguagens compiladas. Geralmente essas linguagens são bastante portáveis porque basta que o sistema do usuário tenha uma máquina virtual e talvez um conjunto de Frameworks que a suporte. Enquanto o Deploy pode ser mais difícil, já que a quantidade de arquivos ou programas que o computador do usuário deve ter é grande, a manutenção é extremamente simples, tão simples quanto editar ou substituir um arquivo de texto.

Faça testes e alterações nos exemplos deste artigo, crie uma tabela em banco de dados para salvar, editar e abrir seus scripts, crie botões para salvar os scripts em arquivos, com um TSaveDialog. Brinque com o código, e verá o poder que a metaprogramação tem a oferecer para suas aplicações.

O pacote de componentes que estudamos não é o único da área. A suíte de componentes JEDI também possui um interpretador de código, e para quem gosta do gerador de relatórios Fast Report, junto com ele vem não um, mas três interpretadores de código: Pascal, JavaScript e C++.

Mesclando esses recursos de interpretação de código com técnicas de RTTI, você pode criar formulários que podem ser bastante customizados, ou até refeitos, pelos seus próprios usuários. Você pode disponibilizar uma IDE para ele fazer isso, assim como faz o Rave Reports ou Fast Reports.

Esperamos que o recurso de interpretação da própria Delphi Language, ou Object Pascal, associado com tipos genéricos, RTTI e outros recursos avançados do Delphi possam criar o melhor de dois mundos: a flexibilidade das linguagens interpretadas e a robustez e velocidade das linguagens compiladas, e que ajude a salvar aqueles projetos legados que precisem de um mundo de alterações cada vez que vão ser implementados em um novo cliente. ●

Links

RemObjects
http://www.remobjects.com/free.aspx

Download do Pascal Script for Delphi
http://www.remobjects.com/download?id=c703bcd3-61bb-45cb-8fb2-75ce7fa74859

Artigos e documentação sobre o Pascal Script no site da RemObjects
http://devcenter.remobjects.com/articles/?catid={C21718A4-7200-49AE-B7CA-89B302B55429}

SynEdit web site
http://synedit.sourceforge.net

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Programação com a API do Windows

Dicas gerais para tarefas comuns

Do que trata o artigo

Neste artigo abordaremos diversas dicas sobre o registro e a API do Windows. Mostraremos como personalizar janelas e outros controles através da API, como usar e tratar mensagens do sistema operacional através dessas funções e várias outras coisas que elas podem nos proporcionar. Veremos como criar uma entrada no registro para programas que inicializam junto com o sistema e diversas customizações do Windows pouco divulgadas.

Para que serve

Com o conhecimento necessário sobre a API e o Registro e, sabendo onde obter mais informações, o desenvolvedor poderá criar aplicações com interfaces diferentes das janelas padrão do Windows, além de automatizar processos, fazer com que softwares comuniquem-se entre si, conhecer um pouco mais dos mecanismos e "entranhas" do Windows e saber onde aplicar tudo isso na prática.

Em que situação o tema é útil

As dicas que veremos podem muito bem ser usadas em sistemas de Lan Houses ou Cyber cafés, onde certas configurações devem ser automáticas ou feitas pelo próprio software, para que sejam transparentes ao dono do estabelecimento. Nesse mesmo tipo de sistema algumas ações dos usuários devem ser limitadas e tanto a API como o registro possuem funções para isso. Analogamente ao caso das Lan Houses, temos os programas que rodam em "franquias", onde muitas opções do Windows podem ser desabilitadas pelo registro. O desenvolvedor pode aprender muito sobre as peculiaridades e funcionamento interno do Windows. Isso facilita a criação de componentes e plugins para o Delphi. Além disso, interfaces mais bonitas ou divertidas de se usar podem ser obtidas com um bom uso da API.

Vitor Luiz Rubio
vitorrubio@gmail.com
http://www.vitorrubio.blogspot.com
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3.

Resumo do DevMan

A API do Windows é um conjunto de arquivos .dll com funções que nos permitem fazer diversas operações. Desde comunicações com outros programas ou com o próprio Windows a interfaces gráficas e auditorias de segurança por exemplo. Existem funções para os mais diversos objetivos e todos os compiladores e IDEs que criam programas nativos para o Windows usam internamente sua API para desenhar janelas e componentes. Os próprios componentes do Delphi usam muito a API do Windows e você poderá aprender como adicionar novas funcionalidades a componentes através da API. Associado a isso temos o registro do Windows, com uma série de opções avançadas que podemos usar para customizar o próprio sistema operacional. Podemos fazer com que nossa própria aplicação faça essas configurações automaticamente.

Nota

O Windows que usaremos no exemplo será a versão XP Professional, porém, versões superiores do Windows serão compatíveis na maioria das chaves do registro.

A API do Windows é um conjunto de arquivos .dll que fazem parte do sistema operacional, porém, expondo várias funções ao meio externo. Nós podemos usar a API do Windows da mesma forma que usamos qualquer outra dll, porém o Delphi contém uma unit que já tem várias das funções mais usadas mapeadas e declaradas.

Durante nossas considerações sobre a API vamos analisar, em cada caso, um cenário onde sua aplicação seria necessária e depois discutiremos como usá-la. Exceto quando o uso da API for muito simples, caso em que inverteremos a ordem: mostraremos primeiro como usar e depois onde seria aplicável. Vamos expor ao longo do artigo diversas dicas do uso de APIs do Windows para a resolução de diversos problemas. Muitas das coisas que conhecemos como dicas nada mais são do que o uso de funções do Windows, que poderiam ser usadas em qualquer linguagem.

O registro do Windows é um conjunto de vários arquivos, dentre eles system.dat e user.dat, mas não se limitando a esses. As informações de configuração do Windows e de diversos outros softwares são armazenadas no registro em forma de árvore, hierarquicamente, de forma muito semelhante a arquivos e diretórios. O formato como esses dados são armazenados e como o arquivo é "feito" não nos interessará porque a Microsoft providenciou na sua API diversas funções para ler e gravar informações do registro. E o Delphi possui uma classe chamada TRegistry que torna o uso dessas funções ainda mais fácil e mais transparente para o desenvolvedor.

Criando janelas dentro de outras janelas

Dentre os vários modelos de disposição de janelas que temos para nossos sistemas, podemos citar o MDI (Multiple Document Interface), onde todas as janelas de um sistema são filhas de uma janela principal, como era a interface dos documentos do Word antes da versão 2007. Uma outra forma é a conhecida SDI (Single Document Interface), onde cada janela abre em um espaço independente. Muito popular hoje é o padrão TDI (Tabbed Document Interface), onde as janelas são dispostas em abas, assim como o Internet Explorer 7 e 8 e o Mozilla Firefox. Imagine um cenário onde você precise, sem usar forms MDI, criar formulários um dentro do outro em vários níveis. Isso é perfeitamente possível com o uso da função SetParent e a propriedade Handle das janelas.

No Windows todo controle gráfico, até mesmo botões e caixas de texto, são considerados janelas, e toda janela tem um Handle, que é um número inteiro positivo que identifica unicamente uma janela. Todas as funções do Windows que usam janelas usam o Handle. Para nós, isso é transparente porque a VCL e outras bibliotecas do Delphi fazem isso automaticamente por nós sem percebermos.

Vamos simular uma aplicação MDI em vários níveis onde não usaremos os forms MDI do Delphi. Além disso, para podermos criar várias instâncias da mesma janela sem usar as variáveis públicas com o nome do formulário e, para evitar vazamentos de memória, vamos projetar um mecanismo no qual poderemos criar quantos formulários quisermos de cada tipo e estes serão destruídos no momento em que os fecharmos clicando no botão X do formulário.

Crie uma aplicação Delphi Win32 e vamos dar o nome de ExemploSetParent a essa aplicação. Vamos criar três formulários, chamados respectivamente de frmPrimeiro, frmSegundo e frmTerceiro e sendo os nomes das units como uFrmPrimeiro, uFrmSegundo e uFrmTerceiro respectivamente. Coloque em cada um dos formulários componentes diferentes, ao seu gosto, como Labels identificando o nome do formulário.

Assegure-se que o primeiro form seja maior que o segundo, e este maior que o terceiro, apenas para, quando colocarmos os formulários um dentro do outro eles não ocultarem controles do próprio formulário. Depois veremos como usar barras de ferramentas e painéis para limitar o espaço que uma janela filha pode tomar. O primeiro formulário será o formulário padrão de nossa aplicação. Os outros dois podemos dizer que seriam um documento e seu detalhe respectivamente.

Nota do DevMan

Antes de começar a trabalhar com a API do Windows é importante lembrar algumas regras básicas. Por exemplo, como a API foi escrita em C/C++, não existem funções que recebam argumentos por referência, então funções que necessitam modificar valores de variáveis devem receber um ponteiro para essa variável.

O uso de ponteiros nessas funções será muito comum. Será bastante comum também o uso de estruturas, como Records.

Uma outra diferença importante são as strings. Enquanto C e C++ trabalham com vetores de Char terminados em #0 (ou \0), conhecidos por nós como pchars, o Delphi tem dois tipos de strings, as ShortStrings podem ter até 255 caracteres, e são formadas por uma estrutura de "tamanho + 1" bytes, onde o primeiro elemento, de um byte, é um número de 0 a 255 indicando o tamanho da string e o próximo elemento é um vetor de caracteres, podendo ter até 255 elementos. Esse tipo de string não é muito usado no nosso cotidiano. O outro tipo é a string "comum". Essa string é formada por um ponteiro que aponta para uma estrutura que contém, no início, 4 bytes formando um cardinal e depois deles um array dinâmico de caracteres. O tipo string do Delphi pode armazenar até 4294967295 letras, que é o tamanho de um cardinal.

Ainda falando sobre strings, a maioria das funções da API terão duas versões, uma terminada com A, que trabalha com AnsiStrings, strings padrão ASCII onde cada caractere possui apenas um byte, e as terminadas com W, que funcionam com WideStrings, strings cujos caracteres têm dois bytes cada, também chamados de utf-16. Isso existe para compatibilidade entre Windows 9x e versões NT ou superiores. O Delphi detecta automaticamente o tamanho do char no seu sistema operacional e cria um mapeamento que torna o uso dessas APIs transparente. Então os nomes das funções que você encontrar terão as terminações A e W suprimidas. Uma olhada no fonte da unit Windows.pas tira bastante dúvidas e revela muitos "segredos" tanto do Sistema Operacional como do Delphi.

Desde a versão 2009 o Delphi tem suporte a strings Unicode, padrão utf-8, que suporta uma faixa enorme de caracteres de diversas línguas, como chinês, árabe etc. Esse tipo de string tem caracteres que podem ter até 4 bytes. Como a maioria das funções da API do Windows não usa esse tipo de string, não entraremos em detalhes sobre ela.

Vamos colocar no primeiro formulário um botão para abrir o segundo, e no segundo um botão para abrir o terceiro. Seus nomes serão btAbrirSegunda e btAbrirTerceira, respectivamente. O Caption de ambos pode ser simplesmente Abrir. Antes de editarmos o código dos botões vamos editar o código do evento OnClose do segundo e terceiro formulários. No Object Inspector selecione o evento OnClose e digite o seguinte código:

```
Action := caFree;
```

Faça isso tanto no segundo como no terceiro formulário. Isso fará com que, ao se fechar o formulário, seu método Free seja executado automaticamente, sendo assim liberado da memória. Não usaremos a variável pública da unidade do formulário.

Vá em Project > Options e assegure-se de que apenas a frmPrimeira será criada com a execução do programa, sendo as outras duas criadas sob demanda, conforme a **Figura 1**.

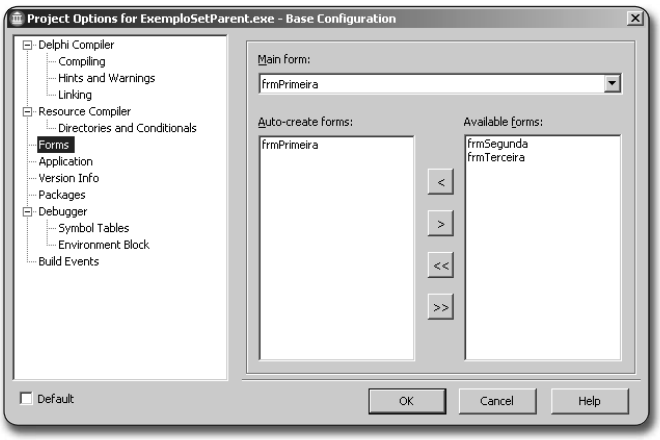


Figura 1. Ordem de criação dos formulários

A unit Windows provavelmente já está declarada no uso de todas as units desse projeto, então não precisaremos nos preocupar. Usaremos o método SetParent dessa unit, mas para não confundir com o método SetParent do form, que tem o mesmo nome, usaremos Windows.SetParent.

Nota

Como dito anteriormente, a VCL usa internamente a API do Windows, então o método SetParent dos forms e outros componentes usam internamente o SetParent da unit Windows, mas de uma forma a facilitar para o desenvolvedor.

Faça com que o primeiro formulário use o segundo, clicando em File > Use unit> uFrmSegundo. Analogamente faça com que o segundo formulário use o terceiro. Vamos dar um duplo clique no botão btAbrirSegunda, do primeiro form, e digitar o código da **Listagem 1**.

Listagem 1. Botão abrir do primeiro formulário

```
procedure TfrmPrimeira.btAbrirSegundaClick(Sender: TObject);
begin
  with TfrmSegunda.Create(Self) do
  begin
    Show;
    Windows.SetParent(handle, Self.Handle);
  end;
end;
```

A função SetParent da unit Windows aceita dois parâmetros, o primeiro é o handle da janela filha e o segundo é o handle da janela que será a mãe. Não devemos confundir esse Parent com os conceitos de Herança, da Programação orientada a objetos, e muito menos com o conceito de Owner dos componentes do Delphi. Esse sistema de Parent das janelas do Windows é apenas uma hierarquia visual.

Colocamos Self como Owner do form que acabamos de criar (segunda) por uma razão especial: podemos criar centenas de "segundos formulários" dentro do primeiro. Se fecharmos todos eles, cada um será destruído automaticamente, porém, se fecharmos o primeiro sem fechar os seus filhos, sendo o primeiro formulário Owner (dono) de todos eles, ele mesmo se encarregará de destruir, automaticamente, todos os seus filhos. Esse é o mecanismo de Owner do Delphi. Poderíamos passar Application como Owner, que também funcionaria, com um pequeno problema: só quando a aplicação inteira fosse fechada é que esses formulários seriam destruídos. Embora isso não faça diferença do primeiro para o segundo, fará diferença do segundo para o terceiro. Por isso decidimos colocar a responsabilidade de cada pai "visual" destruir seus filhos.

Se por um acaso tivéssemos passado nil como Owner, criaríamos um Memory Leak enorme, porque não teríamos referência a nenhuma das dezenas de filhos que criamos, então ao fechar o formulário principal os filhos permaneceriam na memória sem serem destruídos. Esse vazamento iria se revelar apenas quando fechássemos o formulário principal com os filhos abertos. Se fechássemos os filhos antes isso não iria ocorrer, porque cada um ia ser destruído no momento do seu fechamento.

Analogamente, o botão btAbrirTerceira no segundo formulário fará a mesma coisa, mas criando um objeto TFrmTerceira. Veja como ficou a **Listagem 2**.

Listagem 2. Botão abrir do segundo formulário

```
procedure TfrmSegunda.btAbrirTerceiraClick(Sender: TObject);
begin
  with TfrmTerceira.Create(self) do
  begin
    Show;
    Windows.SetParent(handle, self.Handle);
  end;
end;
```

Você pode executar a aplicação e ver como ela se comporta. Pode clicar duas vezes no botão abrir do primeiro formulário, para criar duas TFrmSegunda e, em cada uma delas, clicar três vezes no botão para abrir três TFrmTerceira dentro de cada uma das TFrmSegunda. Pode observar como ficará na **Figura 2**.

Se fecharmos o formulário principal com os filhos e netos abertos, cada um dos "Segundos" destruirá seus três filhos "Terceiros" e logo depois o form principal irá destruir seus dois filhos "Segundos". As propriedades Align dos formulários podem servir para se alinhar os mesmos dentro de seus respectivos pais. Além disso, você pode colocar Toolbars e Panels para facilitar o posicionamento dos formulários e incrementar o visual da sua aplicação. Um Panel pode ser um "pai" de um formulário.

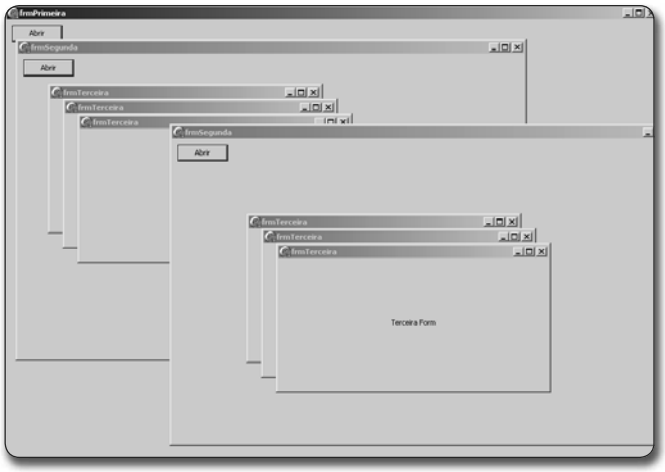


Figura 2. Dois forms dentro do form principal e com três "netas" dentro de cada um

Vamos fazer essas alterações e ver como o sistema se comportará. Coloque no primeiro e no segundo um Panel e uma Toolbar da paleta Win32, conforme a **Figura 3**, e vamos alterar o código conforme a **Listagem 3**. Vamos tirar os botões que havíamos colocado antes e adicionar um botão em cada Toolbar. Passaremos o código das nossas listagens para os botões das barras de ferramentas. Nas barras de ferramentas colocaremos como True a propriedade ShowCaptions. Isso fará com que a propriedade Caption dos botões seja exibida. Deixaremos os botões com o Caption "+". Se o leitor desejar pode usar um Glyph ou ícone para o botão. O desenho dos nossos formulários ficará como na **Figura 3**.

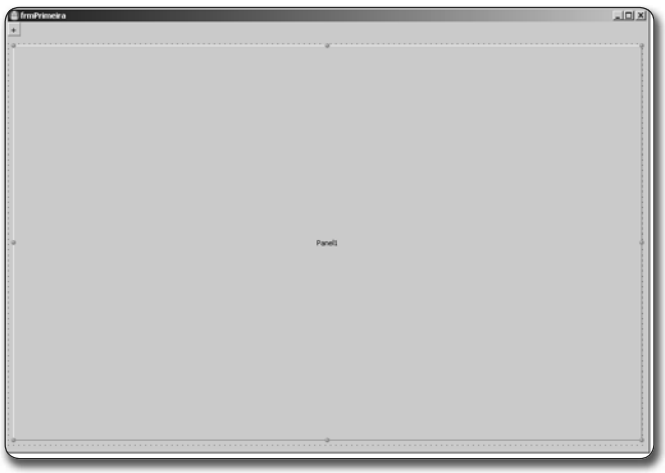


Figura 3. Usando um panel para conter o formulário

Colocamos em cada um dos dois primeiros formulários um Panel e deixamos tanto o nome como o Caption como Panel1. Fizemos uma pequena alteração no código que pode ser vista nas **Listagens 3** e **4**.

A única coisa que modificamos é o handle do Parent. Agora passamos o Panel1 como sendo o pai dos formulários filhos. Isso não muda em nada o mecanismo de destruição dos formulários. Agora a área das janelas filhas está limitada à área dos painéis, conforme vemos na **Figura 4**.

Listagem 3. Usando o Panel como container

```
procedure TfrmPrimeira.ToolButton1Click(Sender: TObject);
begin
  with TfrmSegunda.Create(Self) do
  begin
    Show;
    Windows.SetParent(handle, Self.Panel1.Handle);
  end;
end;
```

Listagem 4. Usando o Panel como container

```
procedure TfrmSegunda.ToolButton1Click(Sender: TObject);
begin
  with TfrmTerceira.Create(Self) do
  begin
    Show;
    Windows.SetParent(handle, Self.Panel1.Handle);
  end;
end;
```

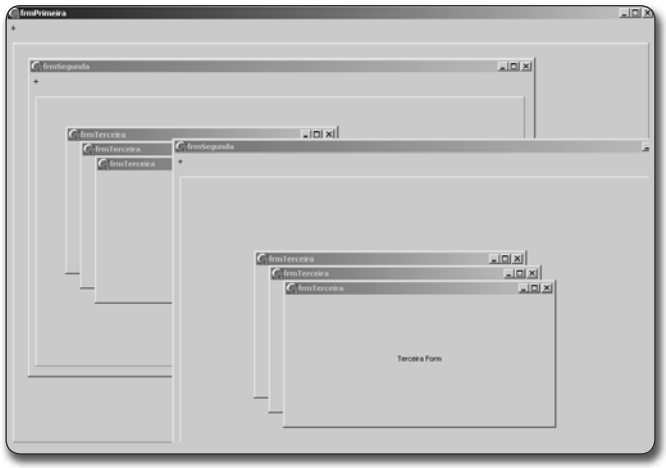


Figura 4. Exibindo janelas abertas dentro de painéis

Mais interessante ainda seria abrir essas janelas dentro de abas para simularmos uma aplicação TDI. A próxima alteração que faremos possibilitará isso, em dois níveis. Vamos substituir nossos Panels por PageControls com Align definido por alCustom e todas as âncoras ativadas.

Nosso código dos botões mudará um pouco. Criaremos dinamicamente um TabSheet para ser uma aba de nosso form, e o Owner desse TabSheet será o PageControl do form "Pai", porém na realidade o TabSheet que será o pai do form. Então cada vez que clicarmos no botão do primeiro formulário aparecerá uma nova aba nele e dentro dessa aba estará uma instância do segundo formulário. E cada vez que clicarmos no botão de um segundo formulário qualquer aparecerá uma nova aba dentro deste, contendo um terceiro formulário.

Assim como o Owner do formulário filho será o formulário pai, o Owner da TabSheet criada dinamicamente será o formulário filho, assim ao se fechar o formulário filho a Tab irá desaparecer porque foi destruída e não ocorrerá nenhum "vazamento" de memória. O código das **Listagens 5** e **6** contém as alterações para isso.

A primeira coisa que fazemos é criar uma variável para a aba e uma para o form. Depois instanciamos o formulário passando Self como Owner, dando assim a responsabilidade de destruí-lo para quem o estiver criando. Analogamente,

ao instanciarmos uma nova aba, passamos como Owner o form que irá contê-la, assim logo que o form for destruído a aba também será. Depois definimos a que PageControl a aba pertence, no caso ao PageContro1 de seu "pai". Colocamos o Caption do formulário criado na própria aba e por último usamos a SetParent para definir a aba como pai do form. A **Figura 5** mostra a janela principal com quatro abas, cada aba contendo uma segunda janela e a última aba contendo outras três abas, cada uma com uma terceira janela.

Figura 5. Aplicação multi-abas

Listagem 5. O segundo form dentro de abas do primeiro

```
procedure TfrmPrimeira.ToolButton1Click(Sender: TObject);
var
  tb: TTabSheet;
  frm: TfrmSegunda;
begin
    frm := TfrmSegunda.Create(self);
  tb := TTabSheet.Create(frm);
  tb.PageControl := self.PageControl1;
  tb.Caption := frm.Caption;
  frm.Show;
  Windows.SetParent(frm.handle, tb.Handle);
end;
```

Listagem 6. Terceiro form dentro de abas do segundo

```
procedure TfrmSegunda.ToolButton1Click(Sender: TObject);
var
  tb: TTabSheet;
  frm: TfrmTerceira;
begin
  frm := TfrmTerceira.Create(self);
  tb := TTabSheet.Create(frm);
  tb.PageControl := self.PageControl1;
  tb.Caption := frm.Caption;
  frm.Show;
  Windows.SetParent(frm.handle, tb.Handle);
end;
```

Acabamos de criar uma aplicação multi-abas que é uma mistura de MDI e TDI. Podemos incrementar ainda mais colocando um botão para fechar as janelas filhas e fazendo com que elas abram maximizadas e sem a barra de títulos. É o que faremos nas **Listagens 7** e **8**.

A única alteração que fizemos foi colocar um botão com o comando Close e acrescentamos essas duas linhas:

```
frm.BorderStyle := bsNone;
frm.Align := alClient;
```

Listagem 7. Modificações para melhorar a aparência do segundo form

```
procedure TfrmPrimeira.ToolButton1Click(Sender: TObject);
var
  tb: TTabSheet;
  frm: TfrmSegunda;
begin
  frm := TfrmSegunda.Create(self);
  tb := TTabSheet.Create(frm);
  tb.PageControl := self.PageControl1;
  tb.Caption := frm.Caption;
  frm.BorderStyle := bsNone;
  frm.Align := alClient;
  frm.Show;
  Windows.SetParent(frm.handle, tb.Handle);
end;
```

Listagem 8. Modificações para melhorar a aparência do terceiro form

```
procedure TfrmSegunda.ToolButton1Click(Sender: TObject);
var
  tb: TTabSheet;
  frm: TfrmTerceira;
begin
  frm := TfrmTerceira.Create(self);
  tb := TTabSheet.Create(frm);
  tb.PageControl := self.PageControl1;
  tb.Caption := frm.Caption;
  frm.BorderStyle := bsNone;
  frm.Align := alClient;
  frm.Show;
  Windows.SetParent(frm.handle, tb.Handle);
end;
```

Essas linhas tirarão a borda e a barra de título do formulário e enquadrarão o mesmo tomando todo o espaço da aba. Agora nossa aplicação parece com um Browser, com a área útil das janelas formando uma aba. Você pode usar algum componente de terceiros que tenha o botão fechar na própria aba ou até criar um. Assim, tornará sua interface ainda mais amigável. Como não é o objetivo desse artigo nos estender muito no assunto de TDI, passaremos ao próximo cenário e veremos a próxima função da API.

Encontrando janelas

Existem alguns cenários onde você precisa encontrar uma janela pelo seu nome, seja para saber se determinado programa está aberto ou até para saber que programas o usuário está abrindo. Componentes de terceiros que são Demos ou Trials usam essa função para saber se o Delphi está aberto, assim o fabricante assegura que o desenvolvedor usará o componente Demo apenas para testes, pois sem o Delphi aberto, no computador de um cliente, por exemplo, o componente torna-se inativo ou emite uma mensagem de alerta.

Para isso usamos a função FindWindow da API. Essa função aceita dois argumentos do tipo Pchar (um vetor de caracteres terminados em #0). O primeiro argumento é o nome da classe e o segundo é o título da janela. Nome da classe é um nome que é igual em todas as janelas do mesmo tipo. Respeitando essa convenção a VCL do Delphi faz com que o nome de classe da janela seja igual ao nome da classe que é o formulário. Um dos dois parâmetros pode ser nil, contanto que o outro possa ser encontrado. Se a função FindWindow encontrar uma janela ela retornará o Handle desta, caso contrário retornará 0.

Crie uma nova aplicação VCL forms Win32 e chame o form principal de frmPrincipal. Coloque alguns botões neste formulário sem se preocupar muito com seus nomes e Captions, pois vamos apenas testar as funções. O primeiro botão conterá

o código da **Listagem 9**, que procurará uma janela cuja classe chama-se TfrmPrincipal e cujo título seja frmPrincipal.

Listagem 9. Exemplo básico de FindWindow

```
procedure TfrmPrincipal.Button1Click(Sender: TObject);
var
  h: THandle;
begin
  h := FindWindow('TfrmPrincipal', 'frmPrincipal');
  ShowMessage(IntToStr(h));
  ShowMessage(IntToStr(Self.Handle));
  if h = self.Handle then
    ShowMessage('encontramos esta janela');
end;
```

Sabendo como encontrar janelas pelo nome podemos colocar janelas de programas abertos pelo usuário dentro de nossas próprias janelas. Por exemplo, a calculadora. O código do próximo botão, na **Listagem 10**, encontrará uma janela chamada Calculadora e encontrando seu handle, usará SetParent para colocar a calculadora dentro da janela principal. A **Figura 6** mostra como a calculadora fica dentro de nossa janela.

Listagem 10. Colocando a calculadora dentro de nossa janela

```
procedure TfrmPrincipal.Button2Click(Sender: TObject);
var
  h: THandle;
begin
  h := FindWindow(nil, 'Calculadora');
  if h <> 0 then
    Windows.SetParent(h, self.Handle);
end;
```

Figura 6. Calculadora do Windows pode ser filha de uma janela qualquer nossa

Antes de executar esse exemplo, abra a calculadora do Windows. A função SetParent tornará nosso form o novo pai da calculadora.

Uma outra utilidade para essa função é saber se o Delphi está aberto. Isso é muito útil tanto para a criação de componentes Trial, quanto na criação de funções de Debug que só serão executadas em tempo de Design ou Debug, apenas pelo desenvolvedor, se o Delphi estiver aberto. Essas funções nunca serão executadas pelo cliente ou usuário final, a não ser que ele também tenha o Delphi.

Sabemos que, no caso do Delphi 2010, uma das janelas é de uma classe chamada TPropertyInspector, que é o Object Inspector, e a janela principal é de uma classe que chama-se TAppBuilder. Caso essas duas janelas existam, significa que o Delphi está aberto e o programa está "em desenvolvimento", caso contrário o Delphi está fechado e o programa está no usuário final.

Criaremos uma função para identificar essas duas janelas conforme a **Listagem 11** e executaremos o teste conforme a **Listagem 12**.

Listagem 11. Função que verifica se o Delphi está aberto

```
function DelphiRodando: boolean;
begin
  Result := (FindWindow('TAppBuilder', nil) > 0) and
  (FindWindow('TPropertyInspector', nil) > 0);
end;
```

Listagem 12. Exemplo de uso da função DelphiRodando

```
procedure TfrmPrincipal.Button3Click(Sender: TObject);
begin
  if DelphiRodando then
    ShowMessage('O delphi está aberto')
  else
    ShowMessage('O delphi não está aberto')
end;
```

Nesta função simplesmente verificamos se as janelas do Object Inspector e a janela principal do Delphi existem, verificando se os handles retornados pela busca das duas são maiores que zero. A função pode ser modificada para trabalhar com versões diferentes do Delphi ou até outras IDEs. Podemos usar essa função para detectar se o Delphi está aberto e saber se podemos ou não, por exemplo, salvar um arquivo de log.

Caso um módulo vendido à parte do seu sistema ou alguma ferramenta auxiliar sejam distribuídos em outros arquivos executáveis, você pode usar FindWindow para verificar se o seu programa principal está aberto e caso não esteja a ferramenta auxiliar não será executada.

Descobrindo a janela ativa e capturando captions de janelas

Ferramentas de log de erro e relatórios de exceções podem e devem apresentar o título da janela ativa por dois motivos, primeiro porque nem sempre a janela ativa que disparou o erro faz parte do seu sistema. Ela pode ter sido aberta por uma DLL ou EXE de terceiros. Outro motivo é que talvez o Caption das janelas mude de acordo com a situação ou módulo aberto, como o Browser, por exemplo, que tem no Caption de sua janela o endereço do site visitado. Além disso, nem sempre o usuário sabe informar com precisão em que janela ele está. Para isso nós usaremos duas funções da API do Windows, a GetForegroundWindow e a GetWindowText.

A função GetForegroundWindow retorna o handle da janela que estiver em foco, ativa, e a função GetWindowText recebe um handle como parâmetro e devolve o título da janela. Para isso, precisaremos usar também a função GetWindowText Length. Lembre-se que a API do Windows foi escrita em C++ e nessa linguagem as strings são diferentes das do Delphi.

Precisamos usar primeiro GetWindowTextLength para obter o tamanho do texto da janela que desejamos para depois alocar memória suficiente em um Pchar, que é um ponteiro para vetor de Char, similar às strings do C++, e depois usar esse mesmo Pchar para armazenar o nome da janela.

Inicie um novo projeto VCL forms e adicione no formulário principal um Memo e um Timer. No evento Timer do Timer1 vamos criar uma variável do tipo THandle para ter o handle da janela em foco, um PChar para conter o título e um Integer para o tamanho. O que vamos fazer é, a cada segundo, verificar a janela em foco e adicionar seu título no Memo. O código da **Listagem 13** é o que vamos estudar.

Listagem 13. Obtendo janela ativa

```
01 procedure Tform1.Timer1Timer(Sender: TObject);
02 var h: THandle;
03   titulo: PChar;
04   tamanho: integer;
05 begin
06   try
07     h := GetForegroundWindow;
08     tamanho := (GetWindowTextLength(h)+1)*sizeof(Char);
       //titulo := AllocMem(tamanho);
       //podem ser usados o GetMem ou AlocMem
09     GetMem(titulo, tamanho);
10     fillchar(titulo^, tamanho, #0);
11     GetWindowText(h, titulo, tamanho);
12     Memo1.Lines.Append( string(titulo) );
13   finally
14     FreeMem(titulo, tamanho);
15   end;
16 end;
```

A Linha 07 captura o handle da janela ativa, nós colocaremos o título em um Pchar que é um ponteiro para caractere e este precisa ser alocado na memória com um certo tamanho, para se comportar como String, e precisa ser inicializado. O tamanho que vamos setar para o título nós podemos obter do próprio Windows, fizemos isso com a Linha 08. Depois de saber o tamanho da String precisamos alocar memória para ela, isso pode ser feito de duas formas:

```
titulo := AllocMem(tamanho);
```

Ou:

```
GetMem(titulo, tamanho);
```

Das duas maneiras a variável título receberá um ponteiro que apontará para uma área já reservada da memória capaz de conter "tamanho" Pchars. É um ponteiro para um vetor. Inicializaremos cada Byte, ou letra, desse vetor com o caractere nulo #0 usando o código da Linha 10.

O uso do operador (^) (Linha 10) aqui é essencial, pois significa "conteúdo de um ponteiro" e nós estamos atribuindo zeros ao conteúdo apontado pelo ponteiro título, e não ao próprio ponteiro. Se esquecermos do operador acabaremos atribuindo zero ao próprio ponteiro e todo ponteiro que aponta para o endereço zero é um ponteiro nulo, que no Delphi é nil.

Com o handle em mãos, o ponteiro título inicializado e sabendo o tamanho do título podemos obter o texto da janela com o método GetWindowText (Linha 11). A conversão do ponteiro Pchar em string é automática, como adicionamos ao memo (Linha 12). No finally liberamos da memória, com FreeMem, o espaço alocado para nosso título (Linha 14).

Faça o teste, execute a aplicação e navegue entre as janelas do Windows, perceberá que os títulos das janelas que você abriu vão aparecendo no memo1. Existe uma outra maneira de fazer a mesma coisa que é com o código da **Listagem 14**. Esse código é muito encontrado em sites de dicas, mas a desvantagem dele é que trabalha com Strings de tamanho fixo, nesse caso 255, usando vetores alocados estaticamente em vez de ponteiros alocados dinamicamente. Se uma janela a ser capturada tiver o tamanho do título maior que 255, parte do texto se perderá.

Listagem 14. Uma alternativa usando vetores de char

```
procedure Tform1.Timer1Timer(Sender: TObject);
var
  h: THandle;
  titulo: array [0..255] of char;
begin
  h := GetForegroundWindow;
  fillchar(titulo, sizeof(titulo), #0);
  GetWindowText(h, titulo, sizeof(titulo));
  Memo1.Lines.Append( string(titulo) );
end;
```

Você também pode usar SetForegroundWindow para definir qual janela será ativa. Isso é útil para uma tela de login de um sistema de Lan House, onde você não pode deixar o usuário clicar em nenhuma tela senão a de login e se ele tirar a janela de login de foco, ela deve voltar a ficar ativa. Comente o código da **Listagem 13** e coloque o seguinte código:

```
SetForegroundWindow(self.Handle);
```

Se você tirar a janela do foco, ela voltará em um segundo, se você diminuir o intervalo do timer esse efeito será bem mais rápido. A janela nunca ficará inativa. Faremos agora um exemplo mais útil, para descobrirmos informações sobre janelas de programas que usamos no nosso cotidiano.

Capturando informações de janelas

Crie uma nova aplicação VCL forms na qual vamos chamar o programa de CapturaJanelas e o form principal de frmCaptura. Deixaremos o form bem pequeno, com aproximadamente 300px de largura e 180px. Configuraremos a propriedade BorderStyle para bsToolWindow e a propriedade formStyle para fsStayOnTop. Isso fará com que a borda e a barra de título da janela sejam bem finas, no estilo de janelas de Toolbox e que a janela seja sempre visível, por cima das outras, não importa que janelas sejam.

Também faremos com que essa janela fique semitransparente, para que possamos ver em parte as janelas que estiverem atrás dela, já que ela ficará na frente de todas. Para isso configure a propriedade AlphaBlend para True e a propriedade AlphaBlendValue para 128. Esse número controla o quanto transparente ou opaco será o form.

Vamos colocar no form oito Labels, quatro mais à esquerda e quatro à direita. Os quatro da esquerda serão apenas títulos e terão seus Captions alterados para Handle, Título, Classe e Tipo respectivamente. Os outros quatro Labels terão os nomes

lblHandle, lblTitulo, lblClasse e lblTipo. Coloque também um Timer no formulário, mas desta vez a propriedade Interval será apenas um milissegundo. Nosso formulário ficará como na **Figura 7**.

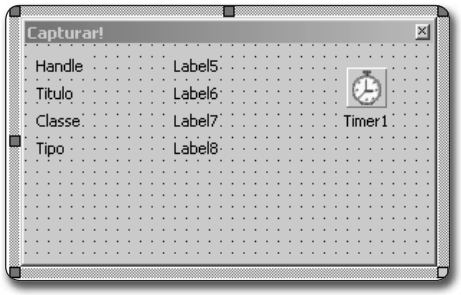


Figura 7. Janela que mostrará informações sobre qualquer janela aberta

O que faremos será capturar o Handle de qualquer janela ou controle que estiver sob o ponteiro do mouse e de posse desse Handle descobrir todas as informações que pudermos sobre essa janela. Para isso usaremos a função GetCursorPos para inicializar uma variável TPoint, que é uma estrutura com as coordenadas x e y, e usaremos WindowFromPoint passando esse TPoint como argumento para obtermos o Handle. Para obter o título da janela faremos a mesma coisa que o exemplo anterior, mas agora obteremos também os nomes da classe e do tipo da janela. Para isso usaremos as funções GetClassName e RealGetWindowClass. O código da **Listagem 15** mostra como podemos fazer isso.

Nota

Ao se trabalhar com janelas na API do Windows é importante lembrar que os termos Classe e Tipo não estão relacionados com Classe da orientação a objetos ou Tipo de dados. Antes, esses termos se referem, respectivamente, ao nome da classe da janela Windows e ao nome do seu tipo. Na documentação da API o Tipo também é chamado de Nome Real da Classe, sendo que a função GetClassName retorna o nome da classe à qual a janela dona do Handle pertence e a função RealGetWindowClass obtém o tipo de janela.

Embora esses termos não estejam relacionados com os nomes das classes da POO, eles podem estar. Um exemplo disso é o fato de o que Delphi quando gera uma janela ou form qualquer criado pelo desenvolvedor, a VCL define como nome de Classe da janela do Windows o mesmo nome da classe do formulário no arquivo.pas, por exemplo, Tform1.

GetCursorPos recebe como parâmetro um TPoint e preenche as coordenadas do mesmo. A função WindowFromPoint recebe o TPoint e devolve o Handle do controle que estiver logo abaixo dele, lembrando que na API, controles, botões e outros tipos de recursos gráficos são considerados janelas também. GetClassName e RealGetWindowClass recebem o Handle obtido, o vetor ou Pchar onde será armazenado o texto e o tamanho desse texto. Depois que obtemos o texto nas variáveis NomeClasse e NomeClasseReal, podemos exibir essas informações nos Labels.

Execute o programa e também a calculadora do Windows. Veja os nomes e Captions das janelas e botões e repare que no caso da calculadora o NomeClasse é diferente do NomeClasseReal. Um uso interessante desse programa é para descobrir se outros programas foram feitos em Delphi ou C++ Builder ou não. Se você encontrar nomes de classes que conhece, como TEdit ou TForm, com certeza esse programa foi feito em Delphi ou C++ Builder.

Listagem 15. Obtendo informações sobre janelas

```
procedure TfrmCaptura.Timer1Timer(Sender: TObject);
var
  hWnd: THandle; //handle da janela
  CrPos: TPoint; //posição do cursor
  NomeJanela: pchar;
  NomeClasse, NomeClasseReal: array [0..255] of char;
  TamanhoNomeJanela: integer;
begin
  NomeJanela := nil;
  GetCursorPos(CrPos); // Identifica a posição do mouse
  hWnd := WindowFromPoint(CrPos); // captura o Handle  na
   posição do cursor
  if hwnd <> 0 then
  try
    lblHandle.Caption := inttostr(hwnd);
    TamanhoNomeJanela := (GetWindowTextLength(hWnd)+1)*sizeof(Char);
    GetMem(NomeJanela, TamanhoNomeJanela);
    FillChar(NomeJanela^, TamanhoNomeJanela, #0);
    FillChar(NomeClasse, SizeOf(NomeClasse), #0);
    FillChar(NomeClasseReal, SizeOf(NomeClasseReal), #0);
    GetWindowText(hWnd, NomeJanela, TamanhoNomeJanela);
    //caption ou texto da janela ou controle
    GetClassName(hWnd, NomeClasse, SizeOf(NomeClasse));
    //classe da janela ou controle
    RealGetWindowClass(hwnd, NomeClasseReal, SizeOf(NomeClasseReal));
    //classe real, ou tipo, do controle
    if Assigned(NomeJanela) then
      lblTitulo.Caption := string(NomeJanela);
    lblClasse.Caption := string(NomeClasse);
    lblTipo.Caption := string(NomeClasseReal);
  finally
    if Assigned(NomeJanela)  then
      FreeMem(NomeJanela, TamanhoNomeJanela);
  end;
end;
```

Transparência e outros efeitos para qualquer janela

Agora que aprendemos como obter Handles de qualquer janela pelo nome, podemos fazer qualquer janela ficar semitransparente, ou piscar a barra de títulos, mesmo que não faça parte da nossa aplicação. Isso é essencial para usar quando temos que abrir um executável de terceiros e destacar para o usuário a janela em que ele deve trabalhar, chamando sua atenção para ele. Janelas semitransparentes também são úteis para se monitorar mudanças imediatas em uma janela ou para comparar, por exemplo, dois relatórios. Vamos criar uma aplicação e salvar o projeto como EfeitosVisuais e a unit principal como efeitos.pas. Coloque um TEdit e dois Buttons no formulário, conforme a **Figura 8**.

Para fazer uma janela piscar temos que usar a função FlashWindow que aceita dois parâmetros, o Handle da janela e o parâmetro Invert, booleano. Caso a janela esteja inativa, ela piscará como ativa e piscará novamente como inativa se Invert for igual a False. Caso Invert seja True a janela termina de piscar com o valor inverso do que estava, ou seja, ativa. Caso a janela já esteja ativa o parâmetro Invert terá efeito sobre o ícone da janela na barra de tarefas. A **Listagem 16** mostra como usar a função FlashWindow.

Na **Listagem 16** usamos FindWindow para encontrar uma janela cujo título seja igual ao do digitado no edTitulo e usamos FlashWindow para piscar a janela. Alem do FlashWindow existe a função FlashWindowEx que tem opções especiais. Ela aceita como argumento uma variável do tipo TFlashWInfo, que vamos discutir depois da **Listagem 17**.

Nota do DevMan

Quando você estiver criando um componente e quiser criar um PropertyEditor ou um ComponentEditor especial, mas não sabe de que classe deve herdar, execute o CapturaJanelas, abra o editor de propriedade ou de componente, por exemplo a lista de TFields de um ClientDataset, ou a lista de MenuItens de um PopUpMenu. Você ficará sabendo de que classes que são essas janelas, e usando o localizar do Windows você pode até encontrar os fontes delas entre os arquivos do Delphi. Isso ajudará bastante a escrever ou modificar seus próprios editores de propriedades e componentes. Você pode, por exemplo, criar um editor de SQL personalizado para o SQLDataSet. Da mesma forma você pode saber de que classe são certos plug-ins do Delphi para escrever os seus próprios.

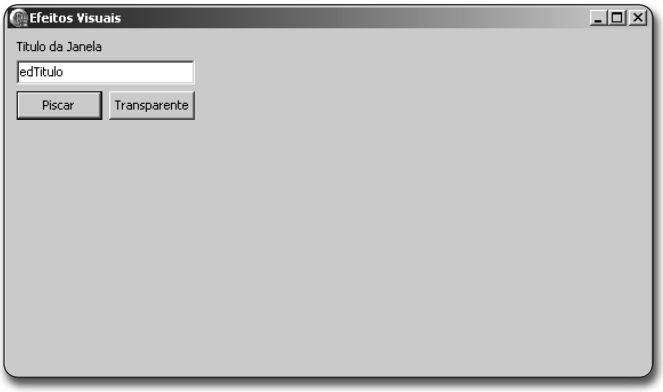


Figura 8. Janela para testes de efeitos visuais

Listagem 16. Piscando a janela com FlashWindow

```
procedure TfrmEfeitos.btPiscarClick(Sender: TObject);
begin
  FlashWindow( FindWindow(nil, pchar(edTitulo.Text)), false );
end;
```

Listagem 17. Piscando a janela com FlashWindowEx

```
procedure TfrmEfeitos.btPiscarClick(Sender: TObject);
var
  fi: TFlashWInfo;
begin
  fi.cbSize := sizeof(fi);
  fi.hwnd := FindWindow(nil, pchar(edTitulo.Text));
  fi.uCount := 10;
  fi.dwTimeout := 0;
  fi.dwFlags := FLASHW_ALL or FLASHW_TIMERNOFG;
  FlashWindowEx(fi);
end;
```

O tipo TFlashWInfo é um Record formado pelos seguintes membros: cbSize, do tipo inteiro, que deve receber o tamanho da própria estrutura TFlashInfo, hwnd é do tipo HWND, ou THandle e recebe um Handle de uma janela, uCount é o número de vezes que piscará, no nosso caso será 10, mas pode ser setado para 0 caso queiramos que pisque continuamente e dwTimeout é a frequência em milissegundos com que a janela piscará. O parâmetro dwFlags é uma série de valores hexadecimais que podem ser combinados booleanamente com o operador or. Os valores possíveis são mostrados na **Tabela 1**.

Parâmetro	Valor	Descrição
FLASHW_ALL	valor = $3	Pisca tanto o título da janela como a barra de tarefas, é o mesmo que usar FLASHW_CAPTION or FLASHW_TRAY flags.
FLASHW_CAPTION	valor = $1	Pisca apenas o título da janela
FLASHW_STOP	valor = $0	Se já estiver piscando, faz parar de piscar e voltar ao estado normal.
FLASHW_TIMER	valor = $4	Pisca continuamente até que se use FLASHW_STOP para parar.
FLASHW_TIMERNOFG	valor = $C	Pisca continuamente até o usuário clicar na janela tornando-a ativa.
FLASHW_TRAY	valor = $2	Pisca o botão na barra de tarefas.

Tabela 1. Valores para o parâmetro dwFlags

Um outro efeito visual interessante que podemos usar é a transparência e vazão. Para isso precisamos conhecer a função GetWindowLong. Esta função aceita como parâmetros o Handle de uma janela e um número inteiro que servirá de índice para a área extra de memória da janela que especifica um dentre vários valores possíveis a serem retornados. Isso porque essa função é bem genérica, ela pode retornar desde um Handle a informações sobre estilos da janela. Existem vários valores possíveis para se obter dessa lista de informações da janela, mas nos concentraremos apenas em GWL_EXSTYLE, que fará com que a função retorne um número inteiro que é o resultado da combinação de todos os estilos visuais da janela.

Analogamente existe a função SetWindowLong que faz o contrário: especificamos um Handle e o índice de uma das informações especiais e modificamos o valor dessa informação especial, em vez de obtermos. Nós devolveremos a esta função, na área de estilos GWL_EXSTYLE, o estilo que obtivemos com a função GetWindowLong combinado com o parâmetro WS_EX_LAYERED, que criará uma camada Alpha no estilo da janela, para efeitos de transparência.

O que estamos fazendo aqui é o mesmo que as propriedades AlphaBlend e AlphaBlendValue dos forms, porém poderemos fazer com qualquer janela aberta. Repare que esses parâmetros GWL_EXSTYLE e WS_EX_LAYERED são constantes inteiras definidas na unit Windows, são flags numéricas.

Depois que definimos que os estilos da nossa janela terão uma camada Alpha, está na hora de usar a função SetLayeredWindowAttributes para definir os atributos dessa camada. Ela recebe quatro argumentos, o primeiro é o Handle da janela e os outros três são atributos da camada Alpha, sendo que o último é formado por dois parâmetros inteiros (flags) combinados com OR. Os atributos são descritos na **Tabela 2**.

Como esse recurso de camadas foi criado no Windows 2000, versões anteriores do Windows 9x ou NT não tem suporte à semi-transparência de janelas. Para facilitar nosso trabalho vamos criar uma função para fazer tudo isso de uma vez, conforme a **Listagem 18**.

Essa função faz aquilo que mencionamos nesta ordem: obtém os estilos visuais da janela combinados em um Integer usando GetWindowLong e adiciona a possibilidade de se trabalhar com uma camada extra usando SetWindowLong

para combinar com "or" os estilos obtidos anteriormente com a flag WS_EX_LAYERED que ativa essa camada. Por último definimos os atributos da camada. Se pularmos essa etapa a janela não terá, visualmente, nenhuma diferença de uma janela sem a flag ativada. Para testar vamos colocar um Shape oval bem grande, de cor verde, na nossa janela (**Figura 9**).

Nome do atributo	Tipo	Descrição
Transparência	Boolean	Identifica se haverá ou não transparência.
Grau de transparência	Integer	Controla se a janela será mais opaca ou mais transparente
Cor Transparente	Boolean	Identifica se haverão cores transparentes, se houver, os elementos dessas cores formarão "buracos" no formulário. Ideal para criar assistentes a partir de gifs animados com um fundo de uma cor setada como transparente.
Cor	Integer	Será a cor transparente propriamente dita.

Tabela 2. Atributos da função SetLayeredWindowAttributes

Listagem 18. Função Transparente

```
procedure Transparente( hTransp: hwnd;
                        SemiTransp, CorTransp: boolean;
                        NivelTransp: byte;
                        NumCorTransp: cardinal);
var
  AStyle: Integer;
  Flags: integer;
begin
  Flags := 0;
  if SemiTransp then
    Flags := (Flags or LWA_ALPHA);
  if CorTransp then
    Flags := (Flags or LWA_COLORKEY);
  AStyle := GetWindowLong(htransp, GWL_EXSTYLE);
  SetWindowLong(htransp, GWL_EXSTYLE, AStyle or WS_EX_LAYERED);
  SetLayeredWindowAttributes(htransp, NumCorTransp, NivelTransp,
   Flags);
end;
```

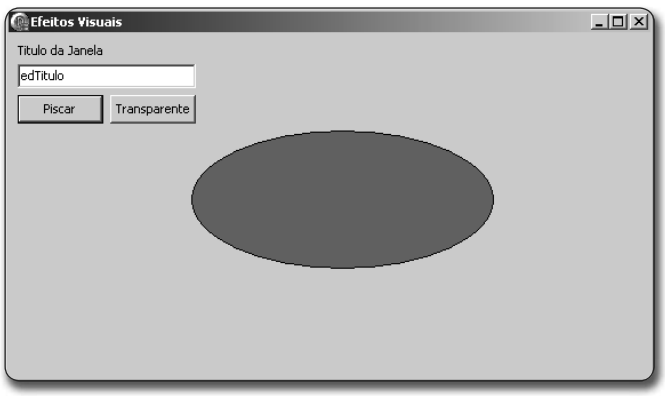


Figura 9. Janela para testes de efeitos visuais com shape oval verde

Vamos usar o código da **Listagem 19** para testar a função. Digitando o próprio título desta janela, "Efeitos Visuais" no Edit e clicando no botão Transparente, o form ficará pouco transparente, já que o valor que usamos (200) é bastante opaco e o Shape verde ficará vazado. O form terá um "buraco" que permite não só ver como clicar no que estiver atrás, conforme podemos ver na **Figura 10**. Muito interessante para Splash Screens e efeitos de animação.

Listagem 19. Evento OnClick do botão btTransparente

```
procedure TfrmEfeitos.btTransparenteClick(Sender: TObject);
begin
  Transparente(FindWindow(nil, pchar(edTitulo.Text)), true,
   true, 200, clGreen);
end;
```

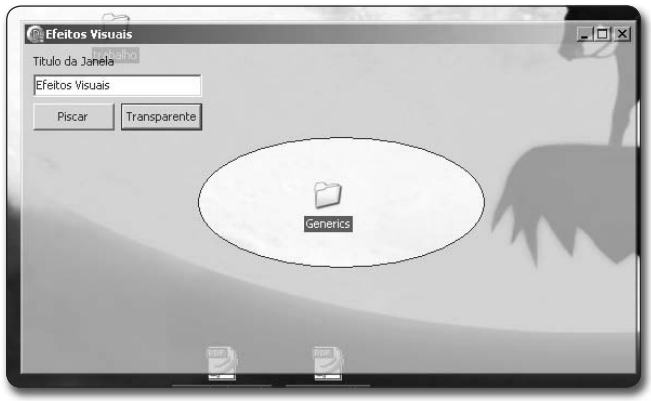


Figura 10. Janela semi transparente no desktop, com um buraco no centro

Mensagens do Windows e comunicação entre executáveis

Uma das funções mais usadas do Windows chama-se SendMessage e ela serve literalmente para um monte de coisas. Isso porque o comportamento da função mudará de acordo com os parâmetros que forem passados a ela e existem vários parâmetros que podem ser passados. A maioria deles são ponteiros para estruturas, de diversos tipos diferentes, e cada estrutura serve para uma coisa diferente. Algumas dessas estruturas possuem ponteiros para outras, ou para funções de Callback, gerando uma cadeia.

Neste exemplo vamos usar a função SendMessage para enviar dados de um programa para outro. Por isso vamos criar dois projetos, um emissor e um receptor. A janela do Emissor terá apenas um botão chamado btEnviar, com Caption Enviar e um Label que conterá o Handle da própria janela. Faremos isso para comparar com o que vai aparecer no receptor e assim saberemos quem enviou a mensagem. O programa Receptor terá o Caption do form Receptor e terá um Memo. Cada mensagem enviada a ele será exibida no Memo, o Handle de quem enviou, e a mensagem.

Caso o Receptor seja fechado, o emissor será também. Ambos podem ser emissores e receptores, mas para inicializar uma "conversa" pelo menos um deve saber o Handle do outro, seja obtendo através do título da janela ou passando como parâmetro para o executável. Quando um dos lados tiver o Handle do outro ele pode passar o seu próprio Handle para a sua contraparte, pela mensagem.

Então, para se trocar dados entre os módulos do seu sistema, se você já conhecia a passagem de parâmetros via linha de comando, multicamadas, TCP/IP, DLLs e BPLs, agora conheceu mais uma maneira. Os dados podem ser compartilhados via API do Windows, inclusive entre programas criados em linguagens diferentes.

As **Figuras 11** e **12** mostram as janelas do Emissor e do Receptor respectivamente. No evento OnClick do botão Enviar

do Emissor criaremos uma variável do tipo TCopyDataStruct. Esta é uma estrutura de dados que armazena um ponteiro para uma variável e o tamanho dessa variável. Como passaremos uma String, o tamanho será o Length da String (número de caracteres) multiplicado pelo número de bytes que um caractere tem. Se a String tem 10 letras e cada letra tem 2 bytes, então o tamanho total será 20 bytes. Usaremos a conversão de string para pchar, com o comando PChar(msg) para transformar nossa String num Pchar. Usaremos FindWindow('TfrmReceptor', 'Receptor') para encontrar o Handle do receptor. Encontrando-o vamos colocar a janela do Receptor na frente das outras, caso contrário fecharemos o Emissor. A **Listagem 20** mostra o código do botão Enviar.

Listagem 20. Enviando mensagem

```
procedure TfrmEmissor.btEnviarClick(Sender: TObject);
var
  CopyData: TCopyDataStruct;
  msg: string;
  HandleReceptor: Thandle;
begin
  msg := 'Hello World Again!';
  label1.Caption := IntToStr(self.Handle);
  CopyData.dwData := 0;
  CopyData.cbData := (Length(msg)*SizeOf(Char))+1;
  CopyData.lpData := pchar(msg);
  HandleReceptor := FindWindow('TfrmReceptor', 'Receptor');
  if (SendMessage(HandleReceptor, WM_COPYDATA, WPARAM
   (self.Handle), LPARAM(@CopyData))
    <> 1) then
  begin
      ShowMessage('Janela não encontrada. Fechando o programa.');
      Application.Terminate;
  end
  else
  begin
      BringWindowToTop(HandleReceptor);
  end;
end;
```

No receptor teremos de criar uma procedure para monitorar a mensagem WM_COPYDATA do Windows. Normalmente, em outras linguagens, o certo seria ter um loop infinito sempre chamando a função GetMessage. Enquanto esta fosse diferente de WM_Quit, o clique no X para fechar, o loop continua. Dentro desse loop teríamos um comando Case para tratar cada uma das mensagens do Windows, que são inúmeras. As mensagens do Windows tratam desde comunicações entre processos a eventos de entrada/saída, mouse, teclado e interrupções.

No Delphi não precisamos implementar esse loop, pois ele já o implementa automaticamente. Cada janela feita pela VCL, nos bastidores, já faz esse loop. Foi criado um mecanismo no compilador do Delphi para tratar procedures ou functions seguidas da palavra chave message como sendo um dos desvios, ou Cases, do loop de mensagens do Windows. Então para se interceptar mensagens do Windows, que são enviadas com SendMessage, usamos procedures ou functions com a palavra chave message, seguida da mensagem que queremos tratar.

No caso do nosso receptor, como ele irá receber dados do emissor, a mensagem a ser tratada é a WM_CopyData, conforme visto a seguir na **Listagem 21**. Um método marcado com a palavra chave message é um método especial de manipulação de mensagens do Windows. O desenvolvedor ou usuário não executará esse método, é o Sistema Operacional quem executará esse método quando ocorrer uma mensagem do tipo que ele trata.

Figura 11. Emissor

Figura 12. Receptor

Listagem 21. Recebendo mensagem

```
type
  TFrmReceptor = class(Tform)
    Memo1: TMemo;
  private
    { Private declarations }
  public
    { Public declarations }
    procedure WMCopyData(var Msg: TWMCopyData); message WM_COPYDATA;
  end;

var
  FrmReceptor: TFrmReceptor;

implementation

{$R *.dfm}

{ TFrmReceptor }

procedure TFrmReceptor.WMCopyData(var Msg: TWMCopyData);
var
  msgRecebida: string;
  i: integer;
begin
  //coloca na string o conteudo da mensagem
  msgRecebida := PChar(Msg.CopyDataStruct.lpData);
  //avisa o chamador que deu tudo certo
  Msg.Result := 1;
  Memo1.Lines.Append('Handle do Emissor: ' + IntToStr(Msg.From) +
  ' Mensagem: ' + msgRecebida);
end;
```

Como passamos uma String, podemos obtê-la de volta através do parâmetro Msg. Esse parâmetro é uma estrutura que contém um cardinal, que é a própria mensagem WM_COPYDATA, cujo valor hexadecimal é $004A, um Handle, que é o Handle do remetente e um ponteiro para a CopyDataStruct, um LongInt chamado Result que será tratado no Emissor. É dentro da estrutura CopyDataStruct que recuperamos a nossa String através do parâmetro lpData. Neste caso ele é um PChar e pode ser facilmente convertido para String.

Podemos também passar um ponteiro para qualquer tipo de Record, assim podemos passar nossos dados de negócio de um executável para outro. Vamos criar em ambos uma estrutura de dados TCliente conforme a **Listagem 22**.

Listagem 22. Record TCliente

```
TCliente = record
  Nome: string[50];
  CPF: string[11];
  Idade: integer;
end;
```

As **Listagens 23** e **24** mostram o emissor enviando um TCliente e o Receptor recebendo-o. Optamos por criar um Record com todos os campos primitivos (ShortStrings de tamanho fixo) por ser mais fácil de recuperar do outro lado. Passando Strings longas e objetos teremos o problema da referência indireta, ponteiros para ponteiros, e muitos Access Violations.

A única coisa que mudou realmente nas listagens foi o uso de um TCliente no lugar de Pchar e o uso do operador @ para obter o endereço de memória de msg. O parâmetro cbData agora tem SizeOf(TCliente) que é o tamanho total de TCliente e o uso do operador ^ para obter de volta o valor TCliente apontado pelo ponteiro lpData. Como lpData é um ponteiro não tipado, temos que fazer a conversão de tipo para obter seus dados.

Listagem 23. Emissor com Record TCliente

```
procedure TfrmEmissor.btEnviarClick(Sender: TObject);
var
  CopyData: TCopyDataStruct;
  msg: TCliente;
  HandleReceptor: Thandle;
begin
  msg.Nome := 'Vitor Rubio';
  msg.CPF := '11111111111';
  msg.Idade := 26;
  label1.Caption := IntToStr(self.Handle);
  CopyData.dwData := 0;
  CopyData.cbData := SizeOf(TCliente);
  CopyData.lpData := @msg;

  HandleReceptor := FindWindow('TfrmReceptor', 'Receptor');
  if (SendMessage(HandleReceptor, WM_COPYDATA, WPARAM(self.
   Handle), LPARAM(@CopyData))
    <> 1) then
  begin
      ShowMessage('Janela não encontrada. Fechando o programa.');
      Application.Terminate;
  end
  else
  begin
      BringWindowToTop(HandleReceptor);
  end;
end;
```

Listagem 24. Receptor com TCliente

```
procedure TFrmReceptor.WMCopyData(var Msg: TWMCopyData);
var
    msgRecebida: TCliente;
    i: integer;
begin
  msgRecebida := TCliente(Msg.CopyDataStruct.lpData^);
  Msg.Result := 1;
  Memo1.Lines.Append('Handle do Emissor: ' + IntToStr(Msg.From) +
    ' Nome: ' + msgRecebida.Nome +
    ' CPF: ' + msgRecebida.CPF +
    ' Idade: ' + inttostr(msgRecebida.Idade)  );
end;
```

Simulando pressionamento de teclas

Uma outra coisa interessante que podemos fazer com a API é simular o pressionamento de teclas. Quando você digita algo no teclado isso ocorre automaticamente, mas você pode simular isso para automatizar processos. E usando isso em conjunto com FindWindow você pode até automatizar o preenchimento de formulários. Usaremos para isso a função PostMessage e a keybd_event. A função PostMessage, assim como a SendMessage, tem um grande número de usos possíveis, dependendo dos argumentos passados a ela. A keybd_event é mais específica para criar eventos de teclado. Crie uma nova aplicação e coloque um Memo e três botões conforme a **Figura 13**.

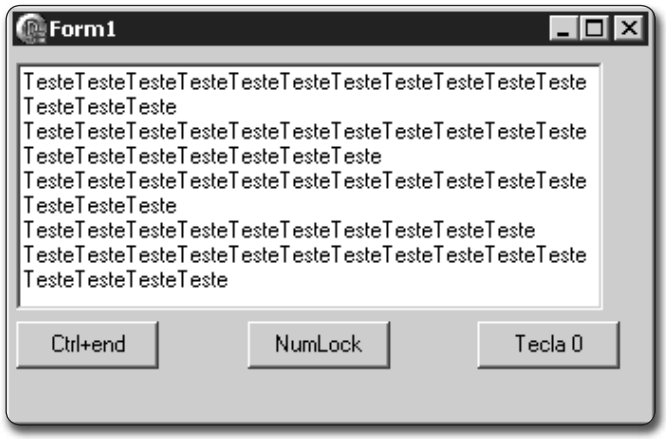


Figura 13. Simulando pressionamento de teclas

A função keybd_event aceita quatro parâmetros. O primeiro é a VirtualKey do Windows, um número inteiro que equivale ao pressionamento de uma tecla. O segundo é também um número inteiro, que tem o código de varredura da tecla. O terceiro é formado pelas flags KEYEVENTF_EXTENDEDKEY, que significa que a tecla é uma tecla estendida ou combinação, e KEYEVENTF_KEYUP que significa que a tecla está sendo solta. Esses parâmetros são combinados com OR. O último parâmetro é um ponteiro para informações extras que não usaremos no nosso exemplo.

Vamos primeiro simular o pressionamento de Ctrl+End. Esse atalho é útil para ir direto para o final de um documento, texto ou último registro de um DataSet, num DbGrid. Vamos ver também como simular o pressionamento da tecla NumLock, para ativá-la ou desativá-la. Por último veremos a simulação da tecla zero do teclado numérico sendo pressionada, usando PostMessage. A linha comentada mostra que poderíamos simular a tecla sendo solta também. O código para simular estas combinações ou teclas encontra-se na **Listagem 25**.

Listagem 25. Simulando Ctrl+End, NumLock e a tecla zero

```
procedure Tform1.Button1Click(Sender: TObject);
begin
  //Ctrl + End
  Memo1.SetFocus;
  keybd_event(VK_CONTROL, 0, KEYEVENTF_EXTENDEDKEY or 0, 0);
  keybd_event(VK_END, 0, 0, 0);
  keybd_event(VK_CONTROL, 0, KEYEVENTF_EXTENDEDKEY or KEYEVENTF_
  KEYUP, 0);
end;

procedure Tform1.Button3Click(Sender: TObject);
begin
  //Num Lock  keybd_event(VK_NUMLOCK, 0, 0, 0);
  keybd_event(VK_NUMLOCK, 0, KEYEVENTF_EXTENDEDKEY or KEYEVENTF_
  KEYUP, 0);
end;

procedure Tform1.Button4Click(Sender: TObject);
begin
  //Tecla 0
  PostMessage(Memo1.Handle,WM_KEYDOWN , VK_NUMPAD0, 0);
end;
```

Primeiro de tudo colocamos o foco no Memo1 para observar o efeito. Executamos keybd_event três vezes, uma passando VK_CONTROL para simularmos a tecla Ctrl e KEYEVENTF_EXTENDEDKEY para dizer que será um evento estendido, por exemplo, uma tecla de atalho ou o pressionamento de duas teclas ao mesmo tempo. A outra execução de keybd_event passa como único parâmetro VK_End, para pressionar o END enquanto VK_Control ainda está sendo pressionado. Por último a função keybd_event, quando passamos a flag KEYEVENTF_KEYUP combinada com KEYEVENTF_EXTENDEDKEY e a virtual key VK_Control, faz com que as duas teclas sejam soltas.

Um comando ou a digitação de uma letra não ocorre no momento do pressionamento da tecla, mas sim no momento que a tecla sobe. Cada tecla pode fazer dois eventos, subir e descer. A tecla NumLock, assim como CapsLock, são teclas especiais que, depois de pressionadas e soltas elas são ativadas. Pressionando e soltando novamente elas são desativadas. Por isso usamos keybd_event duas vezes, uma para pressionar, e outra com o KEYEVENTF_KEYUP, para soltar. Se colocássemos apenas a primeira linha a tecla mudaria de estado uma única vez.

Trabalhando com arquivos

É muito frequente a necessidade de movermos ou copiarmos arquivos ou pastas. A unit ShellApi possui funções importadas das DLLs do sistema operacional que permitem que se façam várias operações do Shell, ou seja, do Windows Explorer. Vamos criar um projeto do zero e chamá-lo de OperaçõesComArquivos. O form principal se chamará frmPrincipal e a unit uPrincipal.pas. Colocaremos dois botões no formulário chamados btCopia1 e btCopia2 com os Captions Cópia 1 e Cópia 2 respectivamente. Criaremos no diretório do projeto três pastas, Origem1, Origem2 e Destino1 e colocaremos alguns arquivos de texto nas pastas Origem1 e Origem2.

Coloque no uses de sua unit principal a unit ShellApi. Dela usaremos a função ShFileOperation. O único argumento que essa função recebe é uma variável do tipo TSHFileOpStruct que é uma estrutura que contém os atributos:

• Wnd*:* HWND – O Handle da janela ou aplicação que ordenou a operação. Caso o arquivo seja grande, ou sejam muitos arquivos, a janela de progresso se tornará modal em relação à janela dona desse Handle;

• wFunc: UINT – Operação, podendo ser FO_MOVE , FO_COPY, FO_DELETE, FO_RENAME. Essas flags tem os valores 1, 2, 3 e 4 respectivamente;

• pFrom*:* PAnsiChar – Nome do arquivo ou diretório de origem;

• pTo*:* PAnsiChar – Nome do arquivo ou diretório de destino. Aplicável somente se não for exclusão;

• fFlags*:* FILEOP_FLAGS – Existem várias flags, que na verdade são números inteiros positivos, ou cardinais, que podem ser combinadas com OR para uma série de efeitos diferentes. Na seção de links há um para a página MSDN, com a documentação das flags usadas nessa estrutura;

• fAnyOperationsAborted*:* BOOL – Não usado;

• hNameMappings*:* Pointer – Não usado;

• lpszProgressTitle*:* PAnsiChar – Permite escolher o título da janela de progresso, somente se a flag FOF_SIMPLEPROGRESS estiver setada.

A **Listagem 26** mostra um exemplo de uso dessa função.

Listagem 26. Copiando arquivos

```
procedure TfrmPrincipal.btCopia1Click(Sender: TObject);
var
  Op: TSHFileOpStruct;
  sde, spara, stitulo: array[0..255] of Char;
  origem, destino, titulo: string;
begin
  origem := ExtractFilePath(Application.ExeName) +
   'Origem1\teste1.txt';
  destino := ExtractFilePath(Application.ExeName) +
   'destino1\teste1.txt';
  titulo := 'Copiando!...';
  FillChar(sde, Sizeof(sde), 0);
  FillChar(spara, Sizeof(spara), 0);
  FillChar(stitulo, Sizeof(stitulo), 0);
  StrPCopy(sde, origem);
  StrPCopy(spara, destino);
  StrPCopy(stitulo, destino);
  with Op do
  begin
    Wnd := Application.Handle;
    wFunc := FO_COPY;
    pFrom := @sde;
    pTo := @spara;
    fFlags := FOF_SIMPLEPROGRESS or FOF_CONFIRMMOUSE or
     FOF_RENAMEONCOLLISION;
    fAnyOperationsAborted := False;
    hNameMappings := nil;
    lpszProgressTitle := @stitulo;
  end;
  ShFileOperation(Op);
end;
```

Como já explicamos a maioria dos membros dessa estrutura, vamos explicar a função ExtractFilePath(Application.ExeName). Application.ExeName tem o nome do arquivo executável de nossa aplicação. ExtractFilePath recupera só o caminho completo, até o último diretório, sem o nome do programa. Isso serve para sabermos onde o executável está e concatenarmos com nossos arquivos de origem e destino.

Combinamos a flag FOF_RENAMEONCOLLISION porque ela é bem interessante: sempre que você copia um arquivo de um diretório para outro e lá no destino já tem um arquivo com o mesmo nome, o Windows sugere a você para substituir os arquivos. Apenas quando você copia arquivos em um mesmo diretório é que o Windows renomeia o arquivo automaticamente, como "cópia de arquivo...". Com o uso desta flag você pode forçar a renomear sempre que houver colisões, mesmo em arquivos

diferentes. Também é possível copiar diretórios. Faremos isso na **Listagem 27** e mostraremos as diferenças a seguir.

Listagem 27. Copiando pastas

```
procedure TfrmPrincipal.btCopia2Click(Sender: TObject);
var
  Op: TSHFileOpStruct;
  sde, spara, stitulo: array[0..255] of Char;
  origem, destino, titulo: string;
begin
  origem := ExtractFilePath(Application.ExeName) + 'Origem2';
  destino := ExtractFilePath(Application.ExeName) + 'Destino2';
  titulo := 'Copiando!...';
  FillChar(sde, Sizeof(sde), 0);
  FillChar(spara, Sizeof(spara), 0);
  FillChar(stitulo, Sizeof(stitulo), 0);
  StrPCopy(sde, origem);
  StrPCopy(spara, destino);
  StrPCopy(stitulo, destino);
  with Op do
  begin
    Wnd := Application.Handle;
    wFunc := FO_COPY;
    pFrom := @sde;
    pTo := @spara;
    fFlags := FOF_SIMPLEPROGRESS or FOF_RENAMEONCOLLISION;
    fAnyOperationsAborted := False;
    hNameMappings := nil;
    lpszProgressTitle := @stitulo;
  end;
  ShFileOperation(Op);
end;
```

As diferenças aqui se resumem a colocar apenas o nome do diretório, sem nome de arquivo. Como o diretório Destino2 não existe a função criará uma cópia da pasta Origem2 com o nome de Destino2. Ou seja, Destino2 terá o mesmo conteúdo de Origem2. Tentando copiar uma segunda vez acontecerá uma cópia de uma pasta Origem2 para dentro de uma outra pasta Destino2, que já existe. Então dentro de Destino2 aparecerá uma Origem2 com o mesmo conteúdo. Só depois da terceira cópia é que ocorrerá a colisão de nomes.

Também é possível usar as funções CopyFile, MoveFile e DeleteFile da unit Windows, que são bem simples de se usar. Um exemplo de CopyFile pode ser visto na **Listagem 28**.

Listagem 28. Copiando arquivo com CopyFile

```
Windows.CopyFile(
  pchar(ExtractFilePath(Application.ExeName) +
   'Origem1\teste1.txt'),
  pchar(ExtractFilePath(Application.ExeName) +
   'Destino1\teste1.txt'),
  true);
```

Esta função não confirma se deseja mesmo fazer a operação ou substituir o arquivo se já existe, mas ele retorna um valor booleano verdadeiro, caso consiga fazer a cópia e falso caso contrário. Os dois primeiros parâmetros são a origem e o destino. O último argumento é usado para saber o que fazer caso o arquivo já exista. Se for passado False o arquivo será sobrescrito. Se for passado True o arquivo não será sobrescrito e a função falhará retornando False.

Uma diferença importante: a função CopyFile não trabalha com "coringas" nos nomes dos arquivos, mas a ShFileOperation sim. Então uma alternativa à **Listagem 26** que copiasse todos os arquivos de texto de um diretório para outro pode ser feita conforme a **Listagem 29**.

Trabalhando com o registro do Windows

Já falamos bastante sobre a API e agora falaremos um pouco sobre o registro do Windows. O registro do Windows é formado por vários arquivos e acessível através de uma série de APIs. O Delphi nos encapsula o uso dessas funções na classe TRegistry, que é muito fácil de usar.

Listagem 29. Alteração necessária para se copiar múltiplos arquivos com coringas

```
Origem  := ExtractFilePath(Application.ExeName) + 'Origem1\*.txt';
destino := ExtractFilePath(Application.ExeName) + 'Destino1\';
```

Inicialmente o registro funciona como uma árvore, igual à estrutura de diretórios do sistema de arquivos de um sistema operacional qualquer. Porém, internamente, o registro se comporta como uma HashTable, ou tabela de pares Chave-Valor.

Para começarmos a aprender sobre o registro vamos criar uma nova aplicação, chamada Registro, e a unit principal será uRegistro.pas. Coloque no seu form alguns botões, arbitrariamente, sem se preocupar com nomes ou Captions. Cada botão executará um dos exemplos que veremos nas próximas listagens. Adicione a unit Registry no uses de seu form principal. Em um botão qualquer crie um objeto da classe TRegistry.

Antes de gravar ou ler algo do registro precisamos definir qual será a RootKey. Dissemos que o registro se comporta como uma árvore, mas na verdade são várias árvores, cada uma partindo de uma de sete raízes diferentes, sendo elas:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_PERFORMANCE_DATA
- HKEY_CURRENT_CONFIG
- HKEY_DYN_DATA

Como não é nosso objetivo documentar o registro detalhadamente, vamos apenas esclarecer que HKEY_CLASSES_ROOT contém todas as classes de componentes COM instalados no sistema e ainda um monte de outras informações, como as associações entre extensões de arquivos e programas que os abrem. É a segunda maior de todas as chaves.

A chave HKEY_DYN_DATA contém apenas dados temporários ou dinâmicos enquanto o computador estiver ligado. Ela é destruída e recriada sempre que o Windows reinicia.

A maior de todas as chaves, e a mais usada é a HKEY_LOCAL_MACHINE. Algumas outras chaves são na verdade atalhos para essa, em nós especiais.

Vamos usar HKEY_LOCAL_MACHINE como raiz do nosso TRegistry. Para isso, faça reg.RootKey := HKEY_LOCAL_MACHINE;

O método OpenKey é muito útil porque combina duas operações. Ele recebe uma String no formato de árvore de diretório, separada por "\", e o segundo parâmetro é um Boolean. Caso seja passado True, a função tenta criar a chave caso ela não exista. O retorno desse método também é um boleano, indicando sucesso ou erro.

Os métodos começados com Write* servem para escrever valores no registro e os começados com Read* servem para ler valores. A **Listagem 30** mostra o primeiro exemplo, onde criaremos uma chave chamada ClubeDelphi dentro da chave

Software e colocaremos uma entrada chamada ValorArmazenado com o valor Hello World.

Atenção

Tome cuidado ao modificar o registro do Windows pois uma falha pode causar o mau funcionamento do S.O.

Nota

Sempre use o registro dentro de um bloco try finally para poder fechar a chave aberta e destruir o objeto TRegistry.

Listagem 30. Iniciando estudos com registro do Windows

```
procedure Tform1.BtGravaClick(Sender: TObject);
var reg: TRegistry;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    if reg.OpenKey('Software\ClubeDelphi', true) then
      reg.WriteString('ValorArmazenado', 'Hello World');
  finally
    reg.CloseKey;
    reg.Free;
  end;
end;
```

Usamos aqui OpenKey para procurar e criar a chave ClubeDelphi e WriteString para escrever na entrada ValorArmazenado o valor Hello World. Para recuperarmos o valor, basta usarmos também OpenKey, que esse método não vai criar a chave novamente caso a encontre. Depois de abrirmos a chave podemos consultar o valor. Nesse momento o uso é idêntico ao de uma estrutura de arquivos INI. A **Listagem 31** mostra como podemos ler o valor.

Listagem 31. Lendo um valor armazenado

```
procedure TfrmRegistro.btLeClick(Sender: TObject);
var reg: TRegistry;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    if reg.OpenKey('Software\ClubeDelphi', true) then
      ShowMessage( reg.ReadString('ValorArmazenado'));
  finally
    reg.CloseKey;
    reg.Free;
  end;
end;
```

Usar o registro é simples conforme exemplificamos. Informações de licença, data de expiração, configurações avançadas do seu software, ou até mesmo Strings de conexão, podem ser armazenadas no registro.

Devemos tomar o cuidado de não alterar ou apagar chaves pertencentes ao Windows ou a outros programas, sempre deletar as chaves criadas por nós ao executar a desinstalação do nosso software (o programa de instalação deve fazer isso automaticamente).

É uma boa prática criar, dentro da chave Software, uma chave com o nome de sua empresa e dentro desta uma chave com o nome de cada um dos seus produtos. Assim cada software de sua empresa pode ter uma configuração independente no computador dos usuários. Configurações globais para todos os produtos podem ficar numa chave a parte ou direto na chave da empresa.

Nota

Da mesma forma que os arquivos, as entradas e chaves de registro também estão sujeitas ao sistema de permissões do Windows. Então um usuário pode ser impedido de criar, ler ou gravar informações no registro.

De agora em diante o que veremos serão dicas de configurações que podemos alterar para mudar opções do Windows e como nosso software pode fazer isso automaticamente.

Desabilitando o Task Manager

Se você deseja que a sua aplicação não seja fechada, ou morta, pelo gerenciador de tarefas do Windows você pode desabilitá-lo pelo registro. Assim, quando o usuário digitar Ctrl+Alt+Del receberá uma mensagem de erro no lugar do gerenciador de tarefas. Essa dica não funciona caso já exista um outro gerenciador de tarefas de terceiros, como o Proccess Explorer, instalado no computador. A **Listagem 32** mostra como desabilitar o Task Manager.

Listagem 32. Desabilitando o Gerenciador de Tarefas

```
01 procedure TfrmRegistro.Button3Click(Sender: TObject);
02 var
03   reg: TRegistry;
04 begin
05   reg := TRegistry.Create;
06   reg.RootKey := HKEY_CURRENT_USER;
07   try
08     if reg.OpenKey('Software\Microsoft\Windows\
       CurrentVersion\Policies\System', True) then
09       reg.WriteInteger('DisableTaskMgr', 1);
10   finally
11     reg.CloseKey;
12     reg.Free;
13   end;
14 end;
```

Para que o gerenciador volte a funcionar você pode substituir o valor 1 da Linha 09 por 0, ou deletar esse valor como o exemplo a seguir:

```
reg.DeleteValue('DisableTaskMgr');
```

Uma outra coisa importante a se notar é que aqui usamos a raiz HKEY_CURRENT_USER. Esta não é uma raiz verdadeira, mas sim um ponteiro, ou atalho, para o lugar certo em HKEY_USERS, que aponta para o usuário corrente (logado). Então a restrição que fizemos ao gerenciador de tarefas vale somente para o usuário corrente. Isso será útil se você precisar desabilitar o gerenciador na inicialização do seu sistema, para evitar que o usuário o mate, ou encerre anormalmente, mas na finalização normal do seu sistema o gerenciador volte a funcionar.

Iniciando junto com o Windows

Existem vários softwares que se inicializam junto com o Windows, por exemplo, MSN, Open Office, Antivírus (e Vírus), Gerenciadores de conexão Wireless entre outros. Isso garante uma certa comodidade e segurança, se a execução do programa for imprescindível para executar tarefas agendadas, receber notificações de eventos em tempo real ou abrir o programa mais rapidamente simplesmente porque ele já estará em memória. Em contraste com isso, muitas pessoas

evitam esses sistemas que se auto-executam na inicialização e retardam a inicialização do Windows, ocupando memória e iniciando mesmo quando não serão usados.

Aprenderemos como incluir nossos programas na inicialização do Windows e como listar e retirar os programas indesejados. Para incluirmos um programa na inicialização do Windows uma das chaves que podemos usar é:

```
Software\Microsoft\Windows\CurrentVersion\Run.
```

Independente do nome da entrada, o valor deverá ser o caminho completo para nosso sistema. Um aplicativo pode se registrar para auto-inicialização no momento da sua instalação, sendo esse registro responsabilidade do programa de instalação, ou então no momento de sua primeira execução. Se o arranque junto com o Windows for opcional, o seu software deverá ter uma seção de configuração onde você possa ativar ou desativar a inicialização junto com o mesmo. A **Listagem 33** mostra como podemos incluir nosso programa na inicialização.

Listagem 33. Registro de inicialização do Windows

```
procedure TfrmRegistro.btInicializacaoClick(Sender: TObject);
var
  reg: TRegistry;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  try
    if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
     Run', true)
    then
      reg.WriteString('Exemplo', Application.ExeName);
  finally
    reg.CloseKey;
    reg.Free;
  end;
end;
```

Aqui usamos a raiz HKEY_CURRENT_USER, que adicionará o sistema na inicialização apenas para o usuário corrente. Configuração similar pode ser feita usando a chave HKEY_LOCAL_MACHINE, válida para todo o computador. Para retirar o seu sistema da inicialização do Windows basta deletar a entrada que acabamos de criar, usando o RegEdit ou usando o método reg.DeleteValue como visto anteriormente.

Outros métodos e propriedades da classe TRegistry são:

- CreateKey – cria uma chave;
- DeleteKey – apaga uma chave;
- DeleteValue – apaga um valor;
- GetKeyNames – obtém uma lista com os nomes das chaves abaixo das chaves correntes;
- HasSubKeys – retorna verdadeiro caso a chave corrente tenha chaves–filhas e falso caso contrário;
- GetValueNames – obtém uma lista com os nomes dos valores contidos na chave corrente;
- KeyExists – verifica se uma chave existe;
- ValueExists – verifica se um valor existe;
- CurrentKey – obtém a chave corrente, em cardinal, uma das constantes da unit Windows;
- CurrentPath – obtém o caminho corrente;
- RootKey – obtém a raiz corrente, em cardinal, uma das constantes da unit Windows;
- RootKeyName – nome da raiz corrente, em String.

A **Listagem 34** mostra um teste completo, com todos esses métodos listados. Logicamente não detalhamos todos os métodos de TRegistry aqui por questões de espaço, mas a classe é bem documentada e não há dificuldade em se encontrar mais informações sobre o assunto. Esses métodos que comentamos são os mais usados.

A **Listagem 34** é um teste com todos os métodos que já explicamos. Criamos uma série de chaves entradas e valores para testar, verificamos sua existência e depois destruímos. A cada método executado invocamos ShowMessage explicando o que foi feito. Isso garante uma pausa a cada método, mesmo que você não esteja debugando. Recomendamos executar esse teste com o RegEdit aberto para verificar cada alteração feita dentro de HKEY_LOCAL_MACHINE\Software\ClubeDelphi. É importante considerar que para referenciar, abrir, criar ou deletar uma chave referindo-se a ela pelo seu caminho absoluto, usamos o método CloseKey para fechar a chave corrente e voltar à raiz. Uma dica interessante é que podemos abrir as chaves do registro uma de cada vez, verificando o caminho completo até o valor armazenado e passando por todo ele. A maneira de se fazer isso pode ser vista na **Listagem 35**.

Poderíamos ter feito todas as verificações unindo as chamadas a OpenKey com AND em vez de fazermos cinco IF's aninhados. Mas com os IF's aninhados você pode colocar um ELSE para cada um e uma mensagem de erro para a falta de cada chave no caminho completo. Por isso, passamos falso no último parâmetro. Assim, caso uma das entradas não exista, somos notificados do erro em vez de tentar criar.

Há diversas configurações que podem ser feitas pelo registro, dependendo do cenário. Por exemplo, se sua empresa utiliza proxy, mas para atualização automática do sistema o proxy deve ser desabilitado para se fazerem os downloads. Nesse caso a chave que deve ser modificada é esta: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings. Modificando-se o valor ProxyEnable para 0 desabilita-se o proxy e 1 habilita-o novamente. Um outro cenário possível seria um sistema que é acessado via Terminal Server, em três servidores diferentes, na mesma empresa, porém a empresa tem apenas uma conexão com a internet que passa por um roteador.

Imagine que o NAT ou encaminhamento das portas já esteja configurado. Como os três não podem atender pela mesma porta, já que externamente na internet o IP visível será o IP do roteador, temos que configurar uma porta diferente para cada um. O computador 1 atenderá pela porta 10001, o 2 pela 10002 e o 3 pela 10003. O próprio programa instalado nos servidores pode mudar a porta do Terminal Server na chave HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Terminal Server\WinStations\RDP-Tcp, valor PortNumber. O valor padrão é 3389, mas você pode mudar esse valor e no Client especificar a porta pela qual quer entrar.

Conclusão

Embora não tenhamos abordado o assunto na sua totalidade, o que vimos aqui pode ser considerado uma boa compilação

Listagem 34. Abrindo uma chave por vez

```
procedure TfrmRegistro.btTesteGeralClick(Sender: TObject);
var
  reg: TRegistry;
  ListaChaves: TStringList;
  ListaValores: TStringList;
begin
  reg := TRegistry.Create;
  ListaChaves := TStringList.Create;
  ListaValores := TStringList.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    reg.CreateKey('Software\ClubeDelphi\Chaveteste');
    ShowMessage('ChaveTeste Criada.');
    reg.CreateKey('Software\ClubeDelphi\Chaveteste\
      ChaveInterna1');
    reg.CreateKey('Software\ClubeDelphi\Chaveteste\
      ChaveInterna2');
    reg.CreateKey('Software\ClubeDelphi\Chaveteste\
      ChaveInterna3');
    ShowMessage('ChaveInterna 1, 2 e 3 Criadas.');
    reg.OpenKey('Software\ClubeDelphi',false);
    ShowMessage('Chave Corrente: ' + reg.RootKeyName);
    ShowMessage('Caminho Corrente: ' + reg.CurrentPath);
    reg.WriteString('NomeEmpresa', 'Banco Developers S.A.');
    ShowMessage('Entrada NomeEmpresa criada com valor ' + reg.
     ReadString('NomeEmpresa'));
    reg.GetValueNames(ListaValores);
    ShowMessage('Lista de valores: ' + ListaValores.Text);
    if reg.ValueExists('NomeEmpresa') then
      ShowMessage('Entrada NomeEmpresa existe')
    else
      ShowMessage('Entrada NomeEmpresa NÃO existe');
    reg.DeleteValue('NomeEmpresa');
    ShowMessage('Entrada NomeEmpresa Excluida.');
    reg.GetValueNames(ListaValores);
    ShowMessage('Lista de valores: ' + ListaValores.Text);
    if reg.ValueExists('NomeEmpresa') then
      ShowMessage('Entrada NomeEmpresa existe')
    else
      ShowMessage('Entrada NomeEmpresa NÃO existe');
    reg.CloseKey;
    reg.OpenKey('Software\ClubeDelphi\Chaveteste',false);

    ShowMessage('Chave Corrente: ' + reg.RootKeyName);
    ShowMessage('Caminho Corrente: ' + reg.CurrentPath);
    reg.GetKeyNames(ListaChaves);
    if reg.HasSubKeys then
      ShowMessage('Existem subchaves, são: ' + ListaChaves.Text);
    if reg.KeyExists('ChaveInterna1') then
      ShowMessage('Chave Existe')
    else
      ShowMessage('Chave NÃO Existe');
    reg.CloseKey;
    reg.DeleteKey('Software\ClubeDelphi\Chaveteste\
     ChaveInterna1');
    ShowMessage('ChaveInterna1 Excluida.');
    reg.GetKeyNames(ListaChaves);
    if reg.HasSubKeys then
      ShowMessage('Existem subchaves, são: ' + ListaChaves.Text);
    if reg.KeyExists('ChaveInterna1') then
      ShowMessage('Chave Existe')
    else
      ShowMessage('Chave NÃO Existe');
  finally
    reg.CloseKey;
    reg.Free;
    ListaChaves.Free;
    ListaValores.Free;
  end;
end;
```

Listagem 35. Abrindo uma chave por vez

```
procedure TfrmRegistro.btUmaPorVezClick(Sender: TObject);
var
  reg: TRegistry;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  try
    if reg.OpenKey('Software', false) then
      if reg.OpenKey('Microsoft', false) then
        if reg.OpenKey('Windows', false) then
          if reg.OpenKey('CurrentVersion', false) then
            if reg.OpenKey('Run', false) then
              reg.WriteString('Exemplo', Application.ExeName);
  finally
    reg.CloseKey;
    reg.Free;
  end;
end;
```

de dicas sobre registro e API que todos os desenvolvedores necessitam durante suas carreiras.

Consideramos como melhorar nossas interfaces, como automatizar processos, copiar arquivos, modificar comportamentos padrões do Windows e fazer configurações automaticamente para que nosso usuário não tenha que por a mão no registro.

Há muitas configurações que ainda poderíamos explorar. Todos esses programas que compramos ou baixamos com configurações avançadas do Windows, nada mais são do que programas editando chaves pouco documentadas do registro.

Quanto à API, exploramos uma parte mínima, sendo que boa parte dos assuntos sequer foi tocada. Por isso é interessante visitar o site MSDN, na seção links, para ver uma documentação completa de tudo o que você pode fazer.

Para dominar os dois assuntos requer-se bastante treinamento e curiosidade, além da leitura de extensos tutoriais e documentações.

Esperamos que esse artigo ajude o leitor a criar sistemas seguros, de qualidade e que impressionem os clientes com sua elegância ou com a facilidade com que o sistema é implantado, bem como conhecimento e automatização de regras e configurações do Windows, que põem a máquina a trabalhar da maneira como o desenvolvedor quiser. ●

Links

Functions – MSDN Library
http://msdn.microsoft.com/en-us/library/dd469351(VS.85).aspx

Windows API
http://msdn.microsoft.com/en-us/library/aa383750(VS.85).aspx

SHFILEOPSTRUCT
http://msdn.microsoft.com/pt-br/library/aa928112.aspx

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Crie um agendador de tarefas OO – Parte 1

Princípios de OO

Do que trata o artigo
Nesta primeira parte você vai aprender como criar um agendador de tarefas utilizando a programação orientada a objetos. Nos próximos artigos desta série serão utilizados padrões de projetos para melhorar o agendador criado neste artigo.

Para que serve
Este artigo mostra os primeiros passos necessários para entender a orientação a objetos. Ao final do artigo você estará apto a classes e manipular os objetos provenientes delas.

Em que situação o tema é útil
Programas orientados a objetos já saem ganhando em relação ao programas com código estruturado, pois quando bem utilizada, esta técnica diminui a quantidade de erros do programa e permite que um sistema possa ser alterado com maior facilidade.

Resumo do DevMan

Nesta série de artigos você vai aprender como utilizar a programação orientada a objetos e entender e aplicar as vantagens dos padrões de projetos. O objetivo desta série é fazer com que você não se limite em apenas conhecer as técnicas existentes - como, por exemplo, herança, polimorfismo etc. - mas que você conheça as vantagens e desvantagens de cada uma, para que possa decidir qual delas terá um melhor resultado para solucionar os diferentes tipos de problemas que aparecem no dia-a-dia.

Rafael Stavarengo
faelsta@gmail.com
Programador-analista da Cheina Informática. Trabalha com Delphi, JME e PHP + ZendFramework. Formando em Tecnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.
http://www.devmedia.com.br/stavarengo

Nesta série você vai aprender pelo menos uma alternativa de solução para problemas como: criar um sistema que possa trocar o banco de dados utilizado sem muito esforço, fazer com que um formulário atualize as informações exibidas na hora em que elas forem alteradas ao invés de colocar um TTimer para ficar verificando se algo mudou ou não. Você vai aprender todos estes conceitos criando um agendador de tarefas parecido com o agendador do Windows. A lógica é simples: o usuário cria uma tarefa, que será agendada para

executar uma ou mais vezes. Neste artigo, que é a primeira parte de uma série, vou apresentar os conceitos mais básicos da orientação a objetos. Não será usado nenhum padrão de projeto nesta primeira parte por dois motivos: por uma questão didática, pois é mais fácil entender padrões de projetos quando você os aplica. E para que os programadores menos experientes com a orientação a objetos adquiriam conhecimento suficientes para entender o restante da série, onde tratarei de designs mais complexos e soluções para simplificá-los. Ao final deste artigo estará pronta a primeira versão do agendador, com um código pouco flexível. É claro que até a versão final o agendador sofrerá muitas mudanças internas comparada com a que será construída aqui. Com esta primeira versão você vai aprender a criar classes e manipular objetos, porém ainda não serão abordadas as técnicas para criar um sistema realmente simples de modificar.

Apresentação da aplicação

Para exemplificar os conceitos da orientação a objetos, vou criar uma aplicação com o objetivo de agendar tarefas para execução. Esta aplicação será parecida com o agendador de tarefas existente no Windows. Primeiro o usuário escolhe algo para ser executado – esta é a tarefa a ser realizada – e depois o usuário escolhe os horários para a execução da tarefa. Em outras palavras, o agendador deve executar um determinado programa em horários pré-programados.

Cada tarefa pode ser agendada para execução uma ou mais vezes, ou seja, é possível agendar a mesma tarefa para executar uma vez no dia 5 e outra no dia 10 de determinado mês. Não deve haver limites para a quantidade de agendamentos de uma tarefa. Se você olhar nas propriedades de alguma tarefa no agendador do Windows, você verá que ele possui tipos diferentes de agendamentos, como: agendamento único, agendamento mensal, agendamento diário, agendamento semanal, entre outros. O agendador que será construído também deve ter tipos diferentes de agendamentos. Para começar vamos criar apenas dois tipos: agendamento único e agendamento diário.

O primeiro passo é identificar quais são os objetos no problema, e cá entre nós: esta é a parte mais fácil. O problema é: deve ser possível agendar a execução de um programa para um ou mais horários. Desta simples frase é possível tirar os dois primeiros objetos da aplicação, são eles: TTarefa e TAgendamento. Veja diagrama da primeira versão destas classes na **Figura 1**.

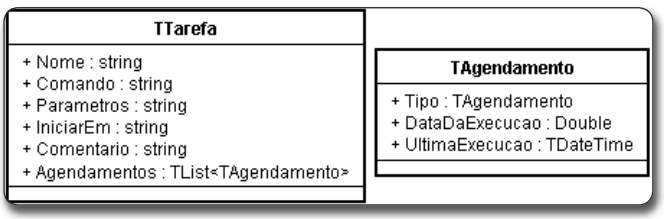


Figura 1. A primeria versão das classes de tarefa e agendamento

A **Figura 1** mostra o quanto as classes TTarefa e TAgendamento são simples. Elas apenas possuem as propriedades que representam as informações que elas gerenciam. Veja a descrição do que significa cada propriedade destas classes a seguir:

- TTarefa:
  - Nome: este é o nome da tarefa – o usuário é quem decide o que vai colocar;
  - Comando: é o caminho do programa a ser executado;
  - Parâmetros: parâmetro para inicialização do programa – caso seja necessário;
  - IniciarEm: diretório em que o programa será iniciado – geralmente é o mesmo diretório onde o programa está instalado;
  - Comentário: alguma informação a mais que o usuário deseje armazenar na tarefa;
  - Agendamentos: lista dos agendamentos para a execução da tarefa.

Nota 3

Se você tem um pouco de intimidade com os objetos, deve ter estranhado estas duas classes. Talvez esteja se perguntando onde foram parar as associações, heranças, encapsulamento e demais técnicas que são tão importantes. Fique tranquilo: as coisas ainda vão mudar muito até a última versão desta aplicação. Todas estas técnicas serão usadas e outras coisas, além de simples atributos, também serão encapsuladas.

- TAgendamento:
  - Tipo: é o tipo de agendamento – por enquanto pode ser agUnico e agDiario;
  - DataDaExecucao: é o momento em que a tarefa deve ser executada – cada tipo de agendamento vai gravar um valor diferente neste atributo: o agendamento agUnico vai armazenar a data e a hora da única execução e o agendamento agDiario vai armazenar apenas a hora em que a tarefa será executada todos os dias; por isto que esta variável é do tipo Double ao invés de TDateTime, pois apesar de armazenar valores de data e hora ou apenas valores de hora eu preferi não me prender ao nome TDateTime ou TTime; na prática isto não faz diferença alguma já que os tipos TDateTime, TTime e TDate na verdade são todos Double;
  - UltimaExecucao: armazena a data da última execução, se é que a tarefa já foi executada alguma vez – uma das funções desta propriedade é ajudar a calcular qual a data da próxima execução da tarefa.

Na **Listagem 1** aparece o código destas duas classes. Você pode ver que não existem segredos ou códigos complexos. Existe apenas a declaração de cada atributo descrito anteriormente, além disto na classe TTarefa existe o método construtor e destrutor (linhas 13 e 14), que serve para criar e destruir a lista de agendamentos da tarefa.

Iniciando o projeto

Após a identificação dos objetos envolvidos no problema e da codificação das classes destes objetos é a hora de começar a criar a interface do programa. Não estou falando da palavra reservada 'interface', estou falando do formulário propriamente dito, formulário este que irá interagir com o usuário. Nesta primeira versão do agendador não existirá um banco de dados, pois este será assunto para um dos próximos artigos desta

série. Veja na **Figura 2** como estão dispostos os componentes dentro do formulário Form1.

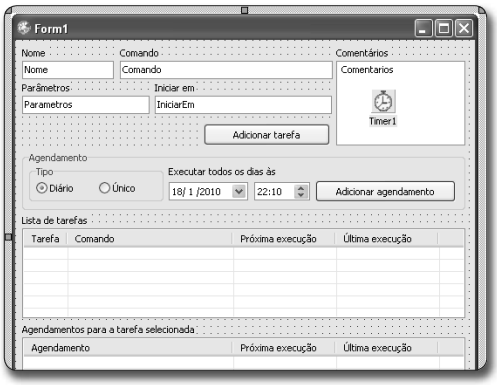


Figura 2. A interface do programa

Para cadastrar e agendar as execuções de uma tarefa através do Form1, o usuário vai informar o nome da tarefa – que será da escolha dele – em seguida irá informar qual o caminho do programa que será executado e se for necessário passar algum parâmetro para o programa durante sua inicialização, estes parâmetros também devem ser informados. Depois de informar todos os dados necessários o usuário deve clicar no botão Adicionar tarefa. A tarefa adicionada vai aparecer no TListView com o TLabel "Lista de tarefas". Para adicionar os agendamentos para a tarefa, primeiro é necessário selecionar a tarefa na lista para que o sistema saiba para qual tarefa os agendamentos serão adicionados. Quando o usuário selecionar uma tarefa na lista exibida no TListView, o botão Adicionar agendamento ficará habilitado e será possível criar agendamentos para a tarefa selecionada.

Para criar agendamentos o usuário escolhe se deseja criar um agendamento para execução diária ou para executar somente uma única vez e em seguida informa o momento em que a tarefa deve ser executada. Se for um agendamento único, deve ser informado o dia e hora em que acontecerá a única execução e se for um agendamento diário, deve ser informada apenas a hora que a tarefa será executada todos os dias. Tenha em mente que o usuário pode criar quantos agendamentos quiser para uma mesma tarefa, indiferente do tipo de agendamento. Ele poderá misturar agendamentos para execução única com agendamentos para execução diária na mesma tarefa.

Agora que você entendeu como a aplicação vai interagir com o usuário, está na hora de dar vida ao Form1 para que ele funcione exatamente como foi descrito nos dois últimos parágrafos. Se você quiser acompanhar a construção desta aplicação – eu aconselho que faça isto – você deve criar uma nova aplicação no Delphi 2010 através do menu File>New>VCL Forms Applications. Em seguida salve a unit do formulário que vai aparecer como Principal.pas e salve o projeto com o nome de Agendador.dpr. Depois você deve criar uma nova unit através do menu File>New>Unit. Salve esta unit com o nome Tarefa.pas. Nesta unit você vai digitar o código que está entre as linhas 1 e 28 da **Listagem 1**. Depois é necessário criar mais uma unit, novamente através do menu File>New>Unit. Salve esta nova unit com o nome Agendamento.pas. Nesta unit você vai colocar o código que está entre as linhas 29 e 40 da **Listagem 1**. No formulário Form1 você deve adicionar e configurar os componentes de acordo com a **Figura 2** (se tiver dificuldade em criar este form, poderá baixar o fonte completo que se encontra disponível no endereço para download deste artigo).

Após criar o projeto de acordo com o que está descrito no último parágrafo, você está pronto para ver a primeira parte de código necessário para dar vida ao Form1. Veja a **Listagem 2**.

Antes de começar a entender o (simples) código da **Listagem 2**, repare bem na linha 6 desta mesma listagem. Nesta linha está declarada a variável Lista. Nesta primeira versão do agendador esta variável será o "banco de dados" da aplicação. Todas as tarefas serão adicionadas nela e consequentemente ao fechar a aplicação, a lista será perdida. Em um dos próximos artigos desta série vou mostrar uma alternativa para criar um sistema que seja independente de banco de dados. Quando isto acontecer esta variável será descartada.

A **Listagem 2** mostra o que possivelmente é o código mais previsível da classe TForm1. Você vê o código responsável por configurar os componentes do formulário Form1. Entre as linhas 8 e 19 está o evento FormCreate. Neste evento, ou seja, quando o agendador é iniciado, a variável Lista que

Nota

Adicione a unit Tarefa.pas e Agendamento.pas na sessão uses da interface da unit Principal.pas.

Listagem 1. Duas humildes classes

```
1  unit Tarefa;
2  Interface
3  uses Generics.Collections, Agendamento;
4  Type
5   TTarefa = class
6   Public
7     Nome: string;
8     Comando: string;
9     IniciarEm: string;
10     Parametros: string;
11     Comentario: string;
12     Agendamentos: TList<TAgendamento>;
13     constructor Create;
14     destructor Destroy; override;
15   end;
16 Implementation
17 uses SysUtils;
18 constructor TTarefa.Create;
19 Begin
20   inherited;
21   Agendamentos := TList<TAgendamento>.Create;
22 end;
23 destructor TTarefa.Destroy;
24 Begin
25   FreeAndNil(Agendamentos);
26   inherited;
27 end;
28 end.
29 unit Agendamento;
30 Interface
31 Type
32   TTipoAgendamento = (agDiario, agUnico);
33   TAgendamento = class
34   Public
35     ÚltimaExecucao: TDateTime;
36     DataDaExecucao: Double;
37     Tipo: TTipoAgendamento;
38   end;
39 Implementation
40 end.
```

Listagem 2. Vida ao Form1: primeira parte de código da classe TForm1

```
1  Implementation
2  uses Generics.Collections, DateUtils, ShellApi, StrUtils;
2  Const
3    AGENDAMENTO_DIARIO = 0;
4    AGENDAMENTO_UNICO = 1;
5  Var
6    Lista: TList<TTarefa>;
7    {$R *.dfm}
8  procedure TForm1.FormCreate(Sender: TObject);
9  Begin
10   Lista := Tlist<TTarefa>.Create;
11   TipoAgendamentoClick(Self);
12   Nome.Clear;
13   Comando.Clear;
14   Parametros.Clear;
15   IniciarEm.Clear;
16   Comentarios.Clear;
17   Data.Date := Today;
18   Hora.Time := Time;
19 end;
20 procedure TForm1.TipoAgendamentoClick(Sender: TObject);
21 Begin
22  Data.Enabled := TipoAgendamento.ItemIndex = AGENDAMENTO_
     UNICO;
23  if Data.Enabled then
24    LabelAgendamento.Caption := 'Executar uma vez em '
25  Else
26    LabelAgendamento.Caption := 'Executar todos os dias às ';
27 end;
```

será usada como banco de dados temporário pela aplicação é criada e, na linha 11 é simulado um clique no TRadioGroup TipoAgendamento. Isto é feito porque alguns dos componentes do formulário são modificados de acordo com a opção selecionada neste TRadioGroup, veja entre as linhas 20 e 27. Se a opção selecionada no TRadioGroup TipoAgendamento for a opção para agendamento único o TDateTimePicker Data é habilitado. Isto acontece porque o agendamento para execução única precisa saber em que dia e hora a tarefa será executada. Consequentemente o TLabel LabelAgendamento também deve ser alterado. Como pode ser visto entre as linhas 23 e 27, se for escolhido o agendamento único, o TLabel vai exibir o texto "Executar uma vez em" e caso seja escolhido o agendamento diário o TLabel vai exibir "Executar todos os dias às".

É para alterar estes componentes que é feita a simulação no TRadioGroup como pode ser visto na linha 11. Além disto, ainda no evento FormCreate existe mais manipulação de componentes. Entre as linhas 12 e 18 os componentes TEdit e TDateTimePicker são iniciados. Como você pode ver é possível afirmar que o evento FormCreate do Form1 inicializa todos os componentes necessários para que o Form1 tenha uma aparência mais elegante para o usuário.

Criar e adicionar uma tarefa

Agora já é possível mostrar a segunda parte de código da classe TForm1. Veja na **Listagem 3**.

Listagem 3. Código para criar e adicionar uma nova tarefa

```
1 procedure TForm1.SalvarTarefaClick(Sender: Tobject);
2 Var
3  Tarefa: Ttarefa;
3 Begin
4  Tarefa := Ttarefa.Create;
5  Tarefa.Comando := Comando.Text;
6  Tarefa.Nome := Nome.Text;
7  Tarefa.Parametros := Parametros.Text;
7  Tarefa.IniciarEm := IniciarEm.Text;
8  Tarefa.Comentario := Comentarios.Text;

9  Lista.Add(Tarefa);

10  MostrarTarefas;
11 end;
```

A **Listagem 3** mostra o código do evento OnClick do TButton SalvarTarefa. Quando o usuário clica neste botão, uma nova tarefa é criada – linha 4 – em seguida os atributos da tarefa criada são alimentados com os valores que o usuário colocou nos TEdits correspondentes – isto acontece entre as linhas 5 e 8. Na linha 9 a nova tarefa é adicionada na variável Lista. Lembre-se que esta variável está simulando o banco de dados da aplicação temporariamente. A declaração desta variável está na linha 6 da **Listagem 2**. Na linha 10 é chamado o método MostrarTarefas que vai listar dentro do TListView ListaDeTarefas todas as tarefas que estão na variável Lista. A implementação deste método está na **Listagem 4**.

Dentro deste método é feita uma iteração com todas as tarefas cadastradas e para cada tarefa encontrada uma nova linha é adicionada no TListView com as informações da tarefa. Para entender o código da **Listagem 4**, vamos direto ao loop que começa na linha 10. Para cada tarefa adicionada na variável Lista, uma nova linha é criada no TListView ListaDeTarefas e em seguida nas linhas 14 e 15 a nova linha criada é preenchida com o nome e o comando da tarefa. Entre as linhas 16 e 20 está um pequeno teste para garantir que mesmo após limpar a lista de tarefas, como pode ver na linha 9, a linha que estava selecionada antes do loop começar ainda estará selecionada quando o loop terminar. A instrução da linha 19 simula o clique do mouse sobre a linha que foi marcada como selecionada na linha 18 – a implementação do método chamado na linha 19 está na **Listagem 5**. Na sequência, existe mais um loop e alguns testes, que apesar de relativamente grande, não chega a ser complexo. As instruções que estão entre as linhas 21 e 30 servem para descobrir qual a data da próxima execução e qual a data da última execução da tarefa. Como cada tarefa pode ter quantos agendamentos o usuário desejar, é necessário iterar com cada agendamento da tarefa para descobrir qual deles tem a maior data de última execução e qual deles tem a menor data de próxima execução que seja maior do que a data atual. Ao final desta lógica simples é possível dizer qual é a próxima data de execução dentre todos os agendamentos da tarefa além de poder dizer também qual a última data de execução dentre todos os agendamentos da tarefa. Com estes valores é possível preencher as duas últimas colunas do TListView ListaDeTarefas, você pode ver isto nas linhas 31 e 33.

Exibir os agendamentos de uma tarefa

O código da **Listagem 4** mostra o método responsável por listar todas as tarefas cadastradas dentro do TListView correspondente. Mas e os agendamentos? Quando serão listados? Os agendamentos são feitos por tarefa, ou seja, cada tarefa possui uma lista interna com seus próprios agendamentos. Por isto para listar os agendamentos primeiro é necessário saber qual tarefa se deseja listar. Isto vai acontecer sempre que o usuário selecionar uma das linhas listadas no TListView ListaDeTarefas. Quando uma linha do TListView é selecionada o evento OnClick é disparado e por isto no evento OnClick do componente ListaDeTarefas será adicionado o código necessário para

exibir a lista de todos os agendamentos da tarefa selecionada no TListView chamado ListaDeAgendamentos. Veja este código na **Listagem 5**.

Entre as linhas 1 e 5 da **Listagem 5** está o código do evento OnClick do TListView ListaDeTarefas. Quando o usuário clica em uma das linhas deste componente o botão AddAgendamento se torna habilitado caso alguma linha realmente esteja selecionada. Isto acontece porque é necessário selecionar a tarefa a qual se deseja adicionar o agendamento. Por isto o botão só fica disponível para ser clicado quando existe uma tarefa selecionada. Por último no evento OnClick, o método MostrarAgendamentos é invocado. Este método irá listar todos os agendamentos da tarefa que foi selecionada. A implementação dele está entre as linhas 6 e 33. Não há segredos neste método. Primeiro é verificado se realmente existe alguma tarefa selecionada para que a lista de agendamento seja exibida. Esta verificação acontece na linha 12. Não se preocupe com o código da função GetTarefaSelecionada, você vai conhecê-lo na **Listagem 7**, assim como vai conhecer na **Listagem 6** o código da função GetDataDaPróximaExecucao que é usada na linha 26. Se o teste da linha 12 retornar verdadeiro, ou seja, se houver uma tarefa selecionada, então é feita a iteração com todos os agendamentos da tarefa e para cada agendamento uma nova linha é adicionada no TListView ListaDeAgendamentos. Em cada linha do componente é adicionada uma descrição para o agendamento, conforme pode ser visto entre as linhas 18 e 25, além de também ser adicionada a data da próxima execução e da última execução, como você pode ver nas linhas 27 e 29.

Nota

Os códigos entre as linhas 12-17 e as linhas 34 e 44 da **Listagem 6** servem apenas para desconsiderar os segundos das datas que serão usadas para o cálculo da próxima execução, pois é mais simples descobrir a data da próxima execução dos agendamentos quando os segundos não são considerados.

Calculando a data da próxima execução do agendamento

Descobrir a data da próxima execução do agendamento não é tão simples quanto somar 2 + 2, pois cada agendamento calcula a data da próxima execução de maneira diferente. Veja **Listagem 6**.

A função declarada na **Listagem 6** recebe como parâmetro uma referência a um objeto do tipo TAgendamento com pode ser visto na linha 1 e o objetivo da função é calcular e retornar o valor da próxima execução do agendamento recebido. Você deve saber que cada tipo de agendamento guarda diferentes valores como data para execução e consequentemente também faz cálculos diferentes para saber quando é a data da próxima execução. O agendamento diário armazena apenas a hora do dia em que deve ser executado, enquanto que o agendamento único guarda o dia e a hora em que a tarefa deve ser executada. Aparentemente a data da próxima execução para um agendamento do tipo único deveria ser apenas a data em que ele foi agendado, simples assim, sem que fosse necessário fazer nenhum teste. Porém não é bem assim. A data da próxima execução do agendamento único realmente é a data em que ele foi agendado, porém, se ele já foi executado, então ele não terá uma próxima execução. Por isto é necessário testar se o agendamento único já foi executado alguma vez e se a data da execução é maior do que a data atual antes de afirmar que a próxima execução será no dia agendado.

Listagem 4. Exibindo as tarefas cadastradas

```
1  procedure TForm1.MostrarTarefas;
2  Var
3   Tr: TTarefa;
4   Ag: TAgendamento;
5   PróximaExecucao, ÚltimaExeucao, Aux: TDateTime;
6   LinhaSelecionada: Integer;
7  Begin
8   LinhaSelecionada := ListaDeTarefas.ItemIndex;
9   ListaDeTarefas.Clear;
10 for Tr in Lista do
11 Begin
12    with ListaDeTarefas.Items.Add do
13    Begin
14      Caption := Tr.Nome;
15      SubItems.Add(Tr.Comando + ' ' + Tr.Parametros);

16      if Index = LinhaSelecionada then
17      Begin
18        Selected := True;
19        ListaDeTarefasClick(Self);
20      end;
21      PróximaExecucao := 0;
22      ÚltimaExeucao := 0;
23      for Ag in Tr.Agendamentos do
24      Begin
25        Aux := GetDataDaPróximaExecucao(Ag);
26        if (PróximaExecucao = 0) or ((Aux > 0) and
          (Aux < PróximaExecucao)) then
27          PróximaExecucao := Aux;

28        if Ag.ÚltimaExecucao > ÚltimaExeucao then
29          ÚltimaExeucao := Ag.ÚltimaExecucao;
30      end;
31      SubItems.Add(
32        IfThen(PróximaExecucao = 0, '<nunca>', DateTimeToStr
          (PróximaExecucao)));
33      SubItems.Add(
34        IfThen(ÚltimaExeucao = 0, '<nunca>', DateTimeToStr
           (ÚltimaExeucao)));
35    end;
36 end;
37 End;
```

Listagem 5. Listar os agendamentos da tarefa selecionada

```
1  procedure TForm1.ListaDeTarefasClick(Sender: TObject);
2  Begin
3   AddAgendamento.Enabled := GetTarefaSelecinada <> nil;
4   MostrarAgendamentos;
5  end;
6  procedure TForm1.MostrarAgendamentos;
7  Var
8   Ag: Tagendamento;
9   Aux: TDateTime;
10 Begin
11  ListaDeAgendamentos.Clear;
12  if GetTarefaSelecinada <> nil then
13  Begin
14    for Ag in GetTarefaSelecinada.Agendamentos do
15    Begin
16      with ListaDeAgendamentos.Items.Add do
17      Begin
18        case Ag.Tipo of
19          agDiario:
20            Caption := 'Todos os dias às ' +
21              FormatDateTime ('HH:mm', Ag.DataDaExecucao);
22          agUnico:
23            Caption := 'Somente uma vez em ' +
24              FormatDateTime('dd-mm-yy "às" HH:mm',
                Ag.DataDaExecucao);
25        end;
26        Aux := GetDataDaPróximaExecucao(Ag);
27        SubItems.Add(IfThen(Aux = 0 , '<nunca', DateTimeToStr(Aux)));
28        Aux := Ag.ÚltimaExecucao;
29        SubItems.Add(IfThen(Aux = 0 , '<nunca', DateTimeToStr(Aux)));
30      end;
31    end;
32  end;
33 end;
```

Listagem 6. Gerando a data da próxima execução do agendamento

```
1  function TForm1.GetDataDaPróximaExecucao(Ag: TAgendamento):
   TDateTime;
2  Var
3   HoraDaExecucao, HoraAtualSemSegundos: TTime;
4   HoraDaÚltimaExecucao: TTime;
5   DataDaExecucao, DataAtualSemSegundos: TTime;
6   DataDaÚltimaExecucao: TTime;
7 Begin
8   Result := 0;
9   case Ag.Tipo of
10    agDiario:
11      Begin
12        HoraDaExecucao := EncodeTime(
13          HourOf(Ag.DataDaExecucao), MinuteOf
              (Ag.DataDaExecucao), 0, 0);
14        HoraAtualSemSegundos := EncodeTime(
15          HourOf(Time), MinuteOf(Time), 0, 0);
16        HoraDaÚltimaExecucao := EncodeTime(
17          HourOf(Ag.ÚltimaExecucao), MinuteOf
              (Ag.ÚltimaExecucao), 0, 0);
18        if (HoraDaExecucao >= HoraAtualSemSegundos) and
19          (HoraDaÚltimaExecucao < HoraDaExecucao) then
20        Begin
21          Result := EncodeDateTime(
22            YearOf(Today), MonthOf(Today), DayOf(Today),
23            HourOf(Ag.DataDaExecucao), MinuteOf
                (Ag.DataDaExecucao), 0, 0);
24        End
25        Else
26        Begin
27          Result := EncodeDateTime(
28            YearOf(Tomorrow), MonthOf(Tomorrow),
                DayOf(Tomorrow),
29            HourOf(Ag.DataDaExecucao), MinuteOf
              (Ag.DataDaExecucao), 0, 0);
30        end;
31      end;
32    agUnico:
33      Begin
34        DataDaExecucao := EncodeDateTime(
35          YearOf(Ag.DataDaExecucao), MonthOf
            (Ag.DataDaExecucao),
36          DayOf(Ag.DataDaExecucao), HourOf(Ag.DataDaExecucao),
37          MinuteOf(Ag.DataDaExecucao), 0, 0);
38        DataAtualSemSegundos := EncodeDateTime(
39          YearOf(Now), MonthOf(Now), DayOf(Now), HourOf(Now),
40          MinuteOf(Now), 0, 0);
41        DataDaÚltimaExecucao := EncodeDateTime(
42          YearOf(Ag.ÚltimaExecucao), MonthOf
              (Ag.ÚltimaExecucao),
43          DayOf(Ag.ÚltimaExecucao), HourOf(Ag.ÚltimaExecucao),
44          MinuteOf(Ag.ÚltimaExecucao), 0, 0);
45        if (DataDaÚltimaExecucao = 0) and
46          (DataDaExecucao >= DataAtualSemSegundos) then
47          Result := DataDaExecucao
48        Else
49          Result := 0;
50      end;
51  end;
52 end;
```

Por outro lado, descobrir a data da próxima execução do agendamento diário é mais simples. Só é preciso verificar se a hora em que o agendamento deve ser executado já passou. Caso tenha passado, a próxima execução será no próximo dia na hora programada e se não passou a execução será no mesmo dia na hora programada.

A implementação da função GetTarefaSelecionada que aparece sendo usada em algumas das listagens anteriores é mostrada na **Listagem 7**. O objetivo desta função é retornar o objeto TTarefa correspondente à linha do TListView ListaDeTarefas que foi selecionada. A lógica é muito fácil de entender. Como você já viu, as tarefas são exibidas no TListView exatamente na ordem em que foram cadastradas, isto significa que a linha 0 do TListView estará exibindo as informações da tarefa que está na posição 0 da variável Lista e assim por diante. Por isto, para pegar o objeto correspondente à linha que o usuário selecionou no TListView, basta retornar o objeto da variável Lista que está na mesma posição da linha selecionada no TListView. Veja na linha 3 da **Listagem 7**, que primeiro é feito um teste para garantir que realmente existe uma linha selecionada e se não existir é retornado o valor nil, ou seja, nada é retornado. Por outro lado a instrução da linha 6 só será executada se uma das linhas do TListView estiver selecionada. Ela retorna o objeto da variável Lista que está na mesma posição em que a linha se encontra dentro do TListView.

Listagem 7. Retornando a tarefa selecionada

```
1 function TForm1.GetTarefaSelecinada: Ttarefa;
2 Begin
3  if ListaDeTarefas.ItemIndex = -1 then
4    Result := nil
5  Else
6    Result := Lista[ListaDeTarefas.ItemIndex];
7 end;
```

Criando agendamentos para a tarefa

Falta pouco para que a primeira versão do agendador fique pronta. As partes do código relacionadas ao cadastro das tarefas bem como o algoritmo para listar as tarefas e seus agendamentos já estão prontas. Falta agora implementar os eventos OnClick do TButton AddAgendamento e OnTimer do TTimer que ficará monitorando as tarefas cadastradas procurando por uma tarefa que esteja na hora de ser executada. Veja a **Listagem 8**.

A **Listagem 8** exibe a implementação do evento OnClick do TButton AddAgendamento. Este TButton tem a finalidade de criar e adicionar um agendamento na tarefa selecionada dentro do TListView ListaDeTarefas. Na linha 5 o agendamento é criado e entre as linhas 6 e 18 está o teste para saber qual o tipo de agendamento foi escolhido e programá-lo corretamente. As constantes que você vê nas linhas 7 e 12 estão declaradas nas linhas 3 e 4 da **Listagem 2**. Elas apenas representam o valor da propriedade ItemIndex do TRadioGroup TipoAgendamento. Caso o usuário tenha escolhido o agendamento diário, então o atributo Tipo do agendamento é definido como agDiario e o atributo DataDaExecucao recebe a hora (somente a hora), em que a tarefa deve ser executada todos os dias. Esta parte do código você pode ver nas linhas 9 e 10. Porém se o usuário escolher o agendamento único, então as instruções das linhas 14, 15 e 16 serão executadas. Primeiro o atributo tipo recebe agUnico. Quando o usuário escolhe o agendamento único, um dos objetos TDateTimePicker é usado para que seja informado o dia da execução e o outro é usado para informar a hora – Lembre-se que existem dois componentes TDateTimePicker. Embora o usuário informe estes valores separadamente é necessário juntá-los antes de alterar o atributo DataDeExecução do agendamento único. Para juntá-los facilmente coloquei a hora escolhida no TDateTimePicker Hora na propriedade Time do TDateTimePicker que armazena o dia, assim, ao chamar o valor da propriedade DateTime do TDateTimePicker Data, ele me retorna a data e hora juntas. Veja nas linhas 15 e 16. Por último, na linha 19 o agendamento é adicionado na tarefa que está selecionada TListView e na linha 20 é feita a atualização dos dois TListViews.

Monitorando os agendamentos das tarefas

Agora só falta implementar o evento OnTimer do TTimer que vai monitorar as tarefas para executá-las na hora correta. Veja esta implementação na **Listagem 9**. Entre as linhas 1 e 11 desta listagem está o evento OnTimer. É muito simples. Para cada

Listagem 8. Criando agendamentos para a tarefa selecionada

```
1  procedure TForm1.AddAgendamentoClick(Sender: TObject);
2  Var
3    Ag: TAgendamento;
4  Begin
5    Ag := TAgendamento.Create;
6    case TipoAgendamento.ItemIndex of
7     AGENDAMENTO_DIARIO:
8     Begin
9       Ag.Tipo := agDiario;
10       Ag.DataDaExecucao := Hora.Time;
11    end;
12    AGENDAMENTO_UNICO:
13    Begin
14      Ag.Tipo := agUnico;
15      Data.Time := Hora.Time;
16      Ag.DataDaExecucao := Data.DateTime;
17    end;
18  end;
19  GetTarefaSelecinada.Agendamentos.Add(Ag);
20  MostrarTarefas;
21 end;
```

Listagem 9. Monitorando e executando as tarefas na hora programada

```
1  procedure TForm1.Timer1Timer(Sender: TObject);
2  Var
3   Tarefa: TTarefa;
4   Ag: TAgendamento;
5  Begin
6   for Tarefa in Lista do
7   Begin
8     For Ag in Tarefa.Agendamentos do
9       if EstaNaHoraDeExecutar(Ag) then
10        Executar(Tarefa, Ag);
11 end;

12 end;

13 function TForm1.EstaNaHoraDeExecutar(Ag: TAgendamento):Boolean;
14 Var
15  Hora, Minuto, HoraAtual, MinutoAtual, Aux: Word;
16  PróximaExecucao: TDateTime;
17 Begin
18  Result := False;
19  case Ag.Tipo of
20    agDiario:
21      Begin
22        DecodeTime(Now, HoraAtual, MinutoAtual, Aux, Aux);
23        PróximaExecucao := GetDataDaPróximaExecucao(Ag);
24        DecodeTime(PróximaExecucao, Hora, Minuto, Aux, Aux);

25        Result := (IsToday(PróximaExecucao) and
          (Hora = HoraAtual) and
26            (Minuto = MinutoAtual));
27      end;
28    agUnico:
29      Begin
30        DecodeTime(Now, HoraAtual, MinutoAtual, Aux, Aux);
31        PróximaExecucao := GetDataDaPróximaExecucao(Ag);
32        DecodeTime(PróximaExecucao, Hora, Minuto, Aux, Aux);

33        Result := (IsToday(PróximaExecucao) and
          (Hora = HoraAtual) and
34            (Minuto = MinutoAtual));
35      end;
36  end;
37 end;
```

Listagem 10. Executando uma tarefa

```
1 procedure TForm1.Executar(Tarefa: TTarefa; Ag: TAgendamento);
2 Begin
3  ShellExecute(0, nil, PChar(Tarefa.Comando), PChar
     (Tarefa.Parametros),
4    PChar(Tarefa.IniciarEm), SW_NORMAL);
5  Ag.ÚltimaExecucao := Now;
6  MostrarTarefas;
7 end;
```

tarefa cadastrada é feito uma iteração com todos seus agendamentos e para cada agendamento existente é verificado se está na hora agendada para execução, como você poder ver na linha 9. Se estiver na hora da execução, simplesmente é chamado o método Execute para executar a tarefa. Você verá o código deste método na **Listagem 10**. Antes, entenda um pouco da função EstaNaHoraDeExecutar. Este método recebe um objeto do tipo TAgendamento e como o próprio nome sugere, tem o objetivo de calcular e retornar um valor booleano indicando se chegou na hora a que o agendamento foi programado. Repare entre as linhas 19 e 36 que o cálculo é diferente para cada tipo de agendamento. Em resumo o que este método faz é verificar se está no mesmo dia e na mesma hora/minuto, em que o agendamento único foi programado e verificar se a hora/minuto atual é igual a hora/minuto que o agendamento diário foi agendado. Nestes dois casos é retornado true.

Agora veja a **Listagem 10**. Também é simples. É a função utilizada na linha 10 da **Listagem 9**. A única coisa que deve ser notada é que após executar o agendamento é alterado o valor do atributo ÚltimaExecucao do agendamento e em seguida os dois TListViews são atualizados.

A primeira versão deste agendador chegou ao fim. Pode compilar e executar a aplicação para ver como tudo funciona. Adicione algumas tarefas e crie alguns agendamentos para elas. Se familiarize com o código criado neste artigo e com a maneira de interagir do sistema, pois isto será importante para as próximas versões.

Conclusão

Esta versão do agendador está pronta, porém muitas coisas precisam ser encapsuladas. A classe TForm1 está sobrecarregada ao extremo e toda a lógica da aplicação está embutida nela. Veremos como resolver isso no próximo artigo. ●

Links

Desvendando Generics e Métodos anônimos no Delphi 2009
http://cc.embarcadero.com/Item/26347

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Design Patterns em PHP

Entenda o padrão Abstract Factory construindo um framework de acesso a dados

Lucas Simões Maistro
lucassimoes@univem.edu.br
Analista/Programador do Centro Universitário Eurípides de Marília. É desenvolvedor Delphi desde 1999 e trabalha com Oracle e SQL Server desde 2001. Professor de Pós-Graduação. Graduado em Análise de Sistemas e Pós-Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Raphael Zanon Rodrigues
zanonr@gmail.com
É Analista/Programador do Centro Universitário Eurípides de Marília. Desenvolvedor Delphi. Graduado em Análise de Sistemas e Pós-Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Resumo do DevMan

O artigo aborda um assunto bastante pertinente aos dias atuais. O uso de padrões de projeto no desenvolvimento de software visa aumentar a produtividade da equipe e a flexibilidade da aplicação, tornando-a mais fácil de entender e dar manutenção. Dentre os vários padrões existentes, escolhemos o Abstract Factory para demonstrar o quão útil os padrões podem ser para os desenvolvedores. Para demonstrar como implementar este padrão, iremos desenvolver um Framework que permite o acesso a diferentes gerenciadores de bancos de dados, tais como MySQL e PostgreSQL.

Do que trata o artigo
O artigo apresenta o padrão Abstract Factory por meio do desenvolvimento de um Framework para acesso a dados. Este Framework deverá facilitar o desenvolvimento ou adaptação de aplicações para acesso a mais de uma base de dados.

Para que serve
O padrão Abstract Factory permite a criação de famílias de objetos (relacionados ou dependentes), por meio de uma interface única, sem a necessidade de especificar a classe concreta, sendo amplamente utilizando no mundo da POO.

Em que situação o tema é útil
O Abstract Factory é bastante útil para aplicações que são ou tendem a ser de grande porte, permitindo a criação de famílias de objetos utilizando apenas uma interface. O padrão utilizado no Framework desenvolvido no artigo possibilita a adaptação de aplicações que têm a necessidade de conexão a mais de uma base de dados sem que seja preciso uma grande alteração no código.

Nos dias atuais, o desafio para os desenvolvedores é criar aplicações de fácil manutenção, reutilizáveis, eficientes, seguras e com os prazos geralmente curtos. Para isso, as equipes de desenvolvimento têm baseado suas aplicações em paradigmas de análise, projeto e programação que suportam essas necessidades. O paradigma de orientação a objetos tem ganhado bastante espaço nos últimos tempos, pois proporciona aos desenvolvedores tais soluções. Porém, a especificidade dessas soluções ou alguns objetos podem ser

tão grandes que esses códigos teriam que ser duplicados a cada vez que necessitássemos de sua criação (ou instanciação).

Caso você ainda não tenha passado por nenhuma situação semelhante, imagine que uma aplicação tem a necessidade de acessar bases de dados distintas como, por exemplo, o MySQL e o PostgreSQL. Seria necessário repetir aquela linha de comando criando a conexão passando o conjunto de diretivas necessárias a cada banco todas as vezes que precisássemos acessar a base. Essa já seria uma prática bastante tediosa, mas, para dar uma melhor visão da situação, imagine que depois da aplicação funcionando, se torna necessário alterar a senha ou mesmo o usuário de algum desses bancos? Nesse momento foi estabelecido o caos, onde é preciso procurar em cada arquivo as diretrizes de conexão do banco e alterar, na "unha", cada uma delas.

Padrões de Projeto

Deixando o nosso problema um pouco de lado, vamos definir o que são os padrões de projeto. Na arquitetura de software, padrões de projeto são soluções desenvolvidas e utilizadas anteriormente por especialistas em grande escala para a solução de problemas comuns para um projeto de software. Por serem reutilizadas muitas vezes e em vários projetos, estas soluções se tornam padrões após terem sua eficácia comprovada.

Mas de onde surgiram os padrões de projetos? É possível considerar que a ideia de padrão de projeto surgiu com um engenheiro civil chamado Christopher Alexander que propôs a criação de catálogos para Arquitetura. Em 1978, Christopher lança seu livro com o nome "A Pattern Language" onde são feitas definições de padrão de projeto.

A partir desta definição, profissionais da área de software passaram a utilizar esses princípios para a criação das primeiras documentações de padrões de projetos para serem utilizados como uma espécie de guia para novos desenvolvedores. Em 1995, Eric Gamma, Richard Helm, Ralph Johnson e John Vlissides - (Gang of Four), publicam o livro "Design Patterns: Elements of Reusable Object-Oriented Software", onde os autores descrevem 23 padrões baseados em suas experiências e que, ainda hoje é a principal referência no assunto.

Resumindo, os padrões de projetos trazem alguns benefícios para o desenvolvimento de aplicações tais como, soluções que já foram testadas e aprovadas, aumentam a eficiência na comunicação entre os participantes do projeto, facilitam o desenvolvimento de módulos coerentes, tornando o sistema mais fácil de entender e dar manutenção.

Bem, até esse ponto vimos de uma forma básica, o que são, como surgiram e como o uso dos padrões de projeto melhoram o desenvolvimento da aplicação, mas existem mais dois pontos que os profissionais relacionados ao desenvolvimento de software devem levar em consideração, e que é bastante crucial para o sucesso do projeto. Quando e como devemos utilizar os padrões de projeto?

Nesse momento, o que mais se deve levar em consideração é o bom senso, ou seja, realizar o levantamento dos requisitos e definir o escopo do projeto. Depois dessa primeira fase devemos identificar possíveis deficiências e se o projeto pode ser otimizado. Feito isso, se for o caso, utilizaremos o padrão de projeto que se ajusta melhor à solução para sanar as deficiências levantadas no projeto. A definição de qual padrão de projeto utilizar se torna bem mais fácil se você tiver uma visão global do projeto e de todo seu funcionamento.

O Padrão Abstracty Factory

Nesse artigo, iremos abordar o padrão Abstract Factory, que é um tipo de padrão de Criação. Com o advento do paradigma de orientação a objetos, profissionais da área de software têm a necessidade de simplificar a instanciação de alguns objetos que, dependendo do tamanho da sua aplicação, podem se tornar cada vez mais complexos para ser instanciados. O Abstract Factory permite a criação de famílias de objetos relacionados ou dependentes, por meio de uma interface única e sem a necessidade de especificar a classe concreta.

O princípio base do Abastract Factory é o encapsulamento, e este conceito é a capacidade de se ocultar os detalhes internos de uma classe. O encapsulamento resolve o problema onde campos expostos podem ser manipulados diretamente a partir de um código externo e que leva a dependências indesejáveis que impedem a alteração na implementação, escondendo alguns componentes e permitindo apenas acessos estilizados ao objeto, o que torna a aplicação mais flexível, fácil de modificar e criar novas implementações. Porém, este padrão utiliza também outros conceitos de orientação a objetos como abstração, herança e polimorfismo que serão abordados mais adiante.

Segundo o princípio do padrão Abstract Factory que é instanciar corretamente uma família de objetos de uma forma transparente para a aplicação, podemos por exemplo, construir uma fábrica responsável por criar uma família de objetos para classes de acesso a bases de dados que precisamos, e de uma forma correta. Nesse momento, vemos que o padrão Abstract Factory pode resolver nosso problema sobre uma aplicação acessar mais de uma base de dados, centralizando em um único lugar a linha de comando responsável por conectar-se à base de dados e diminuindo em muito, o impacto na manutenção do código, caso tenha alguma mudança de usuário ou senha da conexão.

Considerações iniciais do projeto

Cada padrão tem uma finalidade específica e em algumas situações um pode complementar o outro. Para este artigo escolhemos o padrão Abstract Factory para demonstrar a importância de se utilizar padrões de projeto no desenvolvimento de softwares.

Construiremos passo a passo um Framework que poderá ser acoplado a qualquer aplicação PHP que tenha a necessidade de acessar os bancos MySql e/ou PostgreSql. Esse Framework fará, de forma padronizada, todo o trabalho que se tem na hora de realizar a conexão com os gerenciadores de banco de dados existentes, porém caso haja a necessidade o próprio leitor será capaz de implementar o acesso a novos gerenciadores, uma vez que esse padrão permitirá essa alteração de uma forma muito simples.

Para entender melhor o que será feito iniciaremos a implementação utilizando a técnica de desenvolvimento orientado

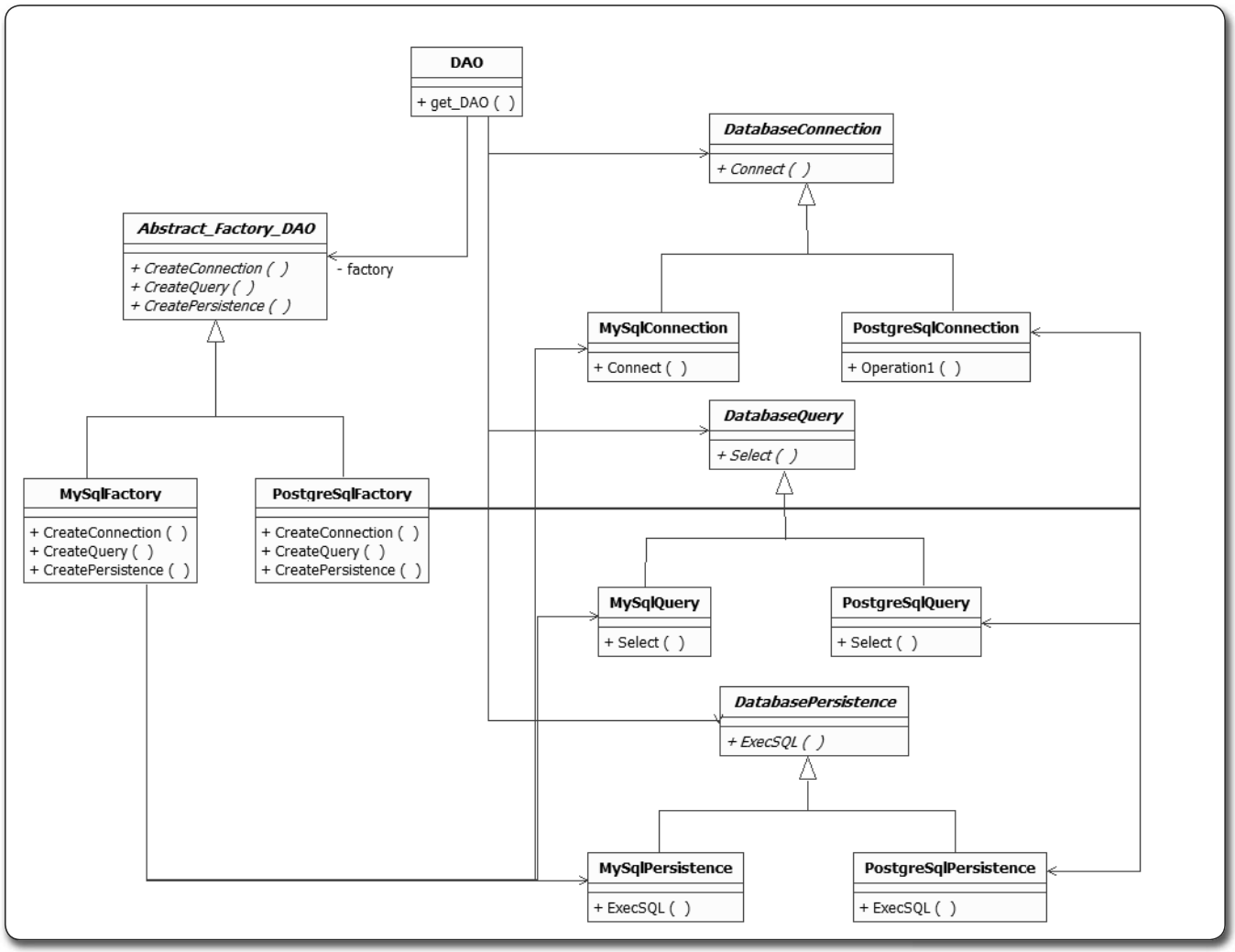


Figura 1. Framework de conexão a dados

a testes, sendo assim criaremos uma classe que utilizará as três funcionalidades esperadas pelo nosso Framework: conexão, consulta e persistência.

Em seguida serão desenvolvidas as nossas classes de Factory, que serão responsáveis por abstrair os métodos do Framework. Essas classes serão responsáveis por direcionar a instanciação dos objetos e execução dos métodos dos produtos de uma determinada família, ou seja, uma vez definido qual banco se quer acessar, a Factory é que faz com que os parâmetros passados cheguem às classes corretas.

Por fim, implementaremos as famílias de produtos e dentro delas o comportamento esperado por uma classe que acesse o banco MySql ou o PostgreSql. Veja na **Figura 1** o diagrama de classe de nosso Framework.

Criando a classe de teste

Conforme foi falado anteriormente utilizaremos a técnica de desenvolvimento dirigido a testes. Essa técnica tem por finalidade automatizar os testes reduzindo dessa forma a quantidade de bugs no software, um dos seus princípios básicos é de que deve-se construir primeiro o mecanismo de teste para em seguida se desenvolver o que deseja, baseado no que o mecanismo de teste espera que este faça.

Para isso, crie um arquivo chamado test.class.php e implemente o código da **Listagem 1**.

A primeira coisa que se nota é que nossa classe fará referência a uma única classe chamada DAO, essa classe deverá fornecer a instância da fábrica concreta, onde estão os métodos de acesso às funcionalidades do Framework, para isso utilizaremos um método chamado get_DAO(). Este deverá receber um parâmetro contendo qual a família de produtos devem ser instanciados, ou seja, qual gerenciador de banco de dados será utilizado, e retornar a instância da fábrica do gerenciador escolhido, neste caso o MySql.

Como os parâmetros usados para se conectar a uma base de dados variam de um gerenciador para o outro, faremos o envio desses parâmetros através de um vetor como pode ser visto entre as Linhas 11 e 15. Ainda na Linha 15 veja que o objeto $conn armazenará a conexão criada a partir dos parâmetros enviados por um método chamado CreateConnection().

Entre as Linhas 17 e 19 testamos a funcionalidade de persistência do framework, veja que primeiro instanciamos uma classe de persistência através do método CreatePersistence(), criamos

uma cláusula SQL de inserção e a executamos na base de dados usando o objeto de persistência que aqui recebe o nome de $insert por meio de um método chamado ExecSQL().

Já nas Linhas 21 e 23 utilizamos o recurso de consulta à base de dados que o Framework nos fornecerá, dessa forma criamos com o método CreateQuery(), a instância da classe de consulta que por sua vez terá um método chamado Select() para realizar uma consulta e devolver um vetor com os dados encontrados.

Após criar a classe de teste é possível ver o quanto nosso framework será flexível e permitirá uma padronização de código de conexão de uma maneira muito interessante para equipes de desenvolvimento. Não faremos neste artigo a classe de teste do PostgreSql, pois entendemos que devido a sua semelhança com o código de teste do MySql seria redundante fazer tal coisa.

Criando o objeto de acesso a dados

A classe DAO é importantíssima para que todo o Framework funcione, pois será sempre a ela que os projetos farão conexão ao Framework e ela determinará o caminho a ser traçado entre o projeto e o banco de dados escolhido a partir da instanciação da fábrica referente ao parâmetro passado. A **Listagem 2** contém a implementação dessa classe, salve-a com o nome DAO.class.php.

Criando a Factory

O padrão Abstract Factory, conforme mencionado anteriormente tem a capacidade de criar famílias de objetos sem que a classe concreta seja especificada, e para que isso seja possível ela explora fortemente a abstração e o polimorfismo, onde a abstração é a capacidade de uma classe representar apenas o domínio do problema e o polimorfismo é quando um método toma diferentes implementações de acordo com sua finalidade.

Sendo assim, para criarmos a chamada fábrica do Framework, iniciaremos criando o arquivo abstract_factory_dao.class.php, este arquivo deve conter o código da **Listagem 3**. Observe que a classe criada é definida como abstrata, ou seja, somente declara os métodos e os parâmetros esperados por eles, porém sua implementação deverá estar nas classes concretas.

Após criarmos a classe abstrata da fábrica devemos implementar seus métodos nas classes concretas MySqlFactory e PostgreSqlFactory, veja nas **Listagens 4** e **5** que os métodos dessas classes são os mesmos da classe Abstract_Factory_DAO, porém em todos eles definimos seu comportamento. Salve as **Listagens 4** e **5** como mysqlFactory.class.php e postgresqlFactory.class.php respectivamente.

É nesse momento que o padrão Abstract Factory entra em ação, pois aqui é feita a referência às classes que representam os produtos da família desejada. O método CreateConnection() recebe os parâmetros necessários para a conexão com seus respectivo banco de dados, instancia a classe de conexão do mesmo e retorna para a classe DAO a conexão com seu status ativo. Já os métodos de consulta e persistência são mais simples, uma vez que apenas devolvem a instância das classes responsáveis por seus comportamentos.

Implementando os produtos abstratos

Os chamados produtos abstratos do padrão Abstract Factory são as classes que abstraem as características básicas daquilo que se espera do Framework, são elas que permitem que métodos com as mesmas características se comportem de maneiras diferentes.

Conforme visto nas **Listagens 6**, **7** e **8** essas classes apenas definem os métodos a serem implementados pelas classes concretas de cada banco de dados, os arquivos dessas classes

Listagem 1. Classe ClientMySQL

```
01 <?php
02 require_once("./DAO.class.php");
03
04 class ClientMySQL {
05  public function __construct() {    }
06
07  public function framework_test() {
08    $dao = new DAO();
09    $mysql = $dao->get_DAO("MySQL");
10
11    $params["server"] = "localhost";
12    $params["user"] = "root";
13    $params["password"] = "";
14    $params["database"] = "academico";
15    $conn = $mysql->CreateConnection($params);
16
17    $insert = $mysql->CreatePersistence();
18    $sql = "INSERT INTO ALUNO (NOME,ENDERECO,BAIRRO,CIDADE,EMAIL)
      VALUES('Lucas','Rua X','Bairro Y','Marília',
       'lucassimoes@univem.edu.br')";
19    $insert->ExecSQL($sql,$conn);
20
21    $consulta = $mysql->CreateQuery();
22    $sql = "SELECT ID_ALUNO, NOME, ENDERECO, BAIRRO, CIDADE,
       EMAIL FROM ALUNO";
23    $array = $consulta->Select($sql,$conn);
24    while ($linha=mysql_fetch_array($array)) {
      echo $linha[0] . " - " . $linha[1] . " - " . $linha[2] .
       " - " .  $linha[3] . " - " . $linha[4] . " - " .
       $linha[5] . "<BR>";
25    }
26  }
27 }
28 ?>
```

Listagem 2. Classe DAO

```
<?php
require_once("./mysqlFactory.class.php");
require_once("./postgresqlFactory.class.php");

class DAO {
 public function __construct() {    }

 public function get_DAO($factory) {
   if ($factory== "MySQL") {
     $db = new  MySQLFactory();
     return $db;
   }
   else if ($factory== "PostgreeSQL") {
     $db = new  PostgreeSQLFactory();
     return $db;
   }
   else
     echo "SGBD não disponível";
 }
}
?>
```

Listagem 3. Classe Abstract_Factory_DAO

```
<?php
 abstract class Abstract_Factory_DAO {

   protected function __construct() {            }

   abstract public function CreateConnection($params);

   abstract public function CreateQuery();

   abstract public function CreatePersistence();
 }
?>
```

Listagem 4. Classe MySqlFactory

```
<?php
 require_once("./abstract_factory_dao.class.php");
 require_once("./mysqlconnection.class.php");
 require_once("./mysqlquery.class.php");
 require_once("./mysqlpersistence.class.php");

 class MySqlFactory extends Abstract_Factory_DAO {
  public function __construct() {
    parent::__construct();
  }

 public function CreateConnection($params) {
  $conn_obj = new MySqlConnection();
  $conn = $conn_obj->Connect($params);
  return $conn;
  }

 public function CreateQuery() {
  $query = new  MySqlQuery();
  return $query;
 }

 public function CreatePersistence() {
  $query = new MySqlPersistence();
  return $query;
  }
 }
?>
```

Listagem 5. Classe PostgreSqlFactory

```
<?php
 require_once("./abstract_factory_dao.class.php");
 require_once("./postgresqlconnection.class.php");
 require_once("./postgresqlquery.class.php");
 require_once("./postgresqlpersistence.class.php");

 class PostgreSqlFactory extends Abstract_Factory_DAO {
  public function __construct() {
  parent::__construct();
 }

 public function CreateConnection($params) {
  $conn_obj = new PostgreSqlConnection();
  $conn = $conn_obj->Connect($params);
  return $conn;

 }

 public function CreateQuery() {
  $query = new  PostgreSqlQuery();
  return $query;

 }

 public function CreatePersistence() {
  $query = new PostgreSqlPersistence();
  return $query;

 }
 }
?>
```

deverão ser salvos como o nome de cada uma delas mais a extensão .class.php.

Finalmente chegamos ao momento de efetivamente implementarmos as funcionalidades do Framework, criaremos agora as classes concretas dos produtos disponíveis.

Classes de Conexão

Nesse instante é que devemos realizar o comportamento esperado na Linha 15 da **Listagem 1**, ou seja, criar uma conexão. Portanto observe que, fazendo o uso dos parâmetros enviados criamos a conexão com o MySql, conforme é visto na **Listagem 9**. O mesmo ocorre na **Listagem 10**, porém, nesse caso, a conexão é feita com o banco de dados PostgreSql. Salve as classes em arquivos com o nome das mesmas.

Listagem 6. Classe DatabaseConnection

```
<?php
 abstract class DatabaseConnection {
 protected function __construct() {            }

 abstract public function Connect($params);
 }
?>
```

Listagem 7. Classe DatabasePersistence

```
<?php
 abstract class DatabasePersistence {
 protected function __construct() {            }

 abstract public function ExecSQL($sql,$conn);
 }
?>
```

Listagem 8. Classe DatabaseQuery

```
<?php
 abstract class DatabaseQuery {
 protected function __construct() {            }

 abstract public function Select($sql,$conn);
 }
?>
```

Classe de Persistência

Na Linha 17 da **Listagem 1** instanciamos através da Factory a classe de persistência do Framework, essas classes realizarão a execução de cláusulas SQL de persistência (INSERT, UPDATE, DELETE). Para isso implementamos o método ExecSQL() que exige como parâmetro, além da instrução, o objeto de conexão com o banco de dados. A única diferença entre as **Listagens 11** e **12** são os comandos de execução de

Listagem 9. Classe MySqlConnection

```
<?php
 require_once("./databaseconnection.class.php");

 class MySqlConnection extends DatabaseConnection {
 public function __construct() {
  parent::__construct();
 }

 public function Connect($params) {

  $conn = mysql_connect($params["server"],$params["user"],
   $params["password"]);
  if( !$conn )
  {
  die("Erro ao conectar no Bando de Dados: ". mysql_error());
  }
  mysql_select_db($params["database"],$conn);
  return $conn;
 }
 }
?>
```

Listagem 10. Classe PostgreSqlConnection

```
<?php
 require_once("./databaseconnection.class.php");

 class PostgreSqlConnection extends DatabaseConnection {
 public function __construct() {
  parent::__construct();
 }

 public function Connect($params) {
  $strconn = "host=".$params["host"]." dbname=".$params
  ["dbname"]." port=".$params["port"]." user=".$params
   ["user"]." password=".$params["password"];
  $conn = pg_connect($strconn);
  if( !$conn )
  {
  die("Erro ao conectar no Bando de Dados: ". pg_last_error());
  }
  return $conn;
  }
 }
?>
```

SQL que devem ser específicos para o banco de dados escolhido. Seguindo o padrão das classes anteriores dê o nome das classes para os arquivos.

Listagem 11. Classe MySqlPersistence

```
<?php
 require_once("./databasepersistence.class.php");

 class MySqlPersistence extends DatabasePersistence {
 public function __construct() {
  parent::__construct();
 }

 public function ExecSQL($sql,$conn) {

    mysql_query($sql,$conn);
 }
 }
?>
```

Listagem 12. Classe PostgreSqlPersistence

```
<?php
 require_once("./databasepersistence.class.php");

 class PostgreSqlPersistence extends DatabasePersistence {
  public function __construct() {
   parent::__construct();
 }

 public function ExecSQL($sql,$conn) {
    pg_query($conn,$sql);
 }
 }
?>
```

Classe de Consulta

Por fim implementaremos as classes concretas de consulta, conforme especificado no arquivo de teste este deve retornar em um vetor a lista de registros encontrados a partir da consulta. Da mesma forma como as classes de persistência, a única diferença entre as **Listagens 13** e **14** são as funções mysql_query() e pg_query(). Após criar essas classes concluímos o desenvolvimento de nosso Framework.

Testando o Framework

Chegou a hora de testar o Framework. Como somente desenvolvemos uma classe de teste para o banco MySql criaremos a base de dados somente nesse gerenciador, mas nada impede que você altere a classe para que essa acesse o PostgreSql ao invés do MySql, somente observe os parâmetros esperados na classe PostgreSqlConnection.

Com o WampServer ativo clique em seu ícone ao lado do relógio e em seguida phpMyAdmin. Dentro da ferramenta escolha o banco de dados MySQL (à esquerda da tela) e após clicar no item SQL digite o script da **Listagem 15**.

Após criar o banco de dados crie um último arquivo chamado index.php, que como todos sabem é interpretado pelo servidor Apache somente indicando a pasta dentro do localhost que contém o projeto (ex.: *http://localhost/Abstract_Factory/*). Lembrando que o diretório do servidor para a hospedagem de seus arquivos encontra-se em C:\Wamp\www\.

Inclua nesse arquivo o conteúdo da **Listagem 16** e em seguida abra o navegador, digite o endereço da aplicação e veja que esta conecta a base de dados, insere um registro e devolve a consulta solicitada ao vetor que posteriormente foi enviado para a tela, conforme pode ser visto na **Figura 2**.

Listagem 13. Classe MySqlQuery

```
<?php
 require_once("./databasequery.class.php");

 class MySqlQuery extends DatabaseQuery {
 public function __construct() {
  parent::__construct();
 }

 public function Select($sql,$conn) {

    $array = mysql_query($sql,$conn);
    return $array;
 }
 }
?>
```

Listagem 14. Classe PostgreSqlQuery

```
<?php
 require_once("./databasequery.class.php");

 class PostgreSqlQuery extends DatabaseQuery {
 public function __construct() {
  parent::__construct();
 }

 public function Select($sql,$conn) {

    $array = pg_query($conn,$sql);
    return $array;
 }
 }
?>
```

Listagem 15. Script do banco de dados MySql

```
CREATE DATABASE academico DEFAULT CHARACTER SET latin1 COLLATE
latin1_swedish_ci;
USE academico;

CREATE TABLE aluno (
  id_aluno int(11) NOT NULL auto_increment,
  nome varchar(400) NOT NULL,
  email varchar(255) default NULL,
  endereco varchar(400) default NULL,
  bairro varchar(150) default NULL,
  cidade varchar(100) default NULL,
  CONSTRAINT PK_ALUNO PRIMARY KEY  (id_aluno)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=27 ;
```

Nota do DevMan

WampServer é um pacote para o Windows que efetua a instalação automática das ferramentas Apache, MySQL e PHP. Quando instalado, o Wamp ativa um servidor Apache e um servidor MySQL, criando um ambiente de testes, permitindo aos desenvolvedores Web criarem aplicações utilizando as ferramentas instaladas.

Para instalar o WampServer, basta efetuar o download da versão mais atual na sessão de downloads do site do desenvolvedor (veja a sessão links), seguir com a instalação não requer nenhum conhecimento adicional e logo após seu término, os scripts PHP já estarão disponíveis.

Vale lembrar que o Wamp utiliza a porta 80 que é de internet HTTP. A questão é que o IIS (Internet Information Services) e outros programas de acesso remoto também podem usar essa porta, sendo assim é necessário manter esses serviços parados enquanto efetuamos nossos testes utilizando o Wamp.

Para o banco de dados PostgreSql realize o download no site oficial (veja a sessão links) e faça a instalação padrão.

Listagem 16. index.php

```
<?php require_once("./test.class.php"); ?>
<html>
  <body onLoad="">
     <center>
     <h2>Framework Test</h2>
     <table border=0>
      <tr><td>
        <?php
         $test = new ClientMySQL();
         $test->framework_test();
         ?>
      </td></tr></table>
     </center>
  </body>
</html>
```

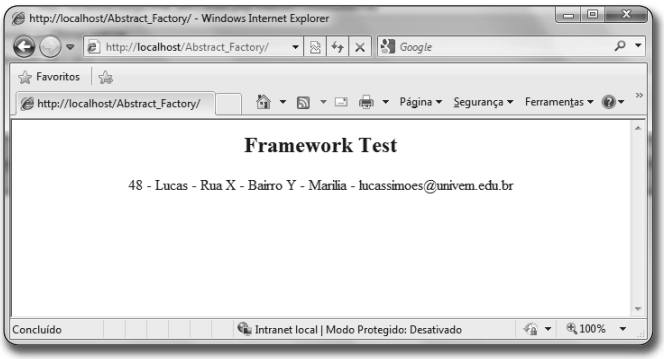


Figura 2. Resultado esperado pela classe de teste

Conclusão

Fica evidente neste artigo que fazer uso de padrões de projeto pode tornar a manutenabilidade de suas aplicações muito mais fácil, já que seguem um padrão, bem como oferece a garantia de que se aplicado na situação correta, problemas em futuras implementações serão minimizados.

O intuito desse artigo foi além de apresentar o padrão Abstract Factory, fornecer ao leitor uma solução para padronizar a implementação de acesso a diversas bases de dados diferentes e estimulá-lo a dar sequência num Framework que certamente trará maior produtividade e flexibilidade no desenvolvimento de softwares em PHP. ●

Links

Site Oficial do WampServer
http://www.wampserver.com/

Site Oficial do PostgreSQL
http://www.postgresql.org/

Twitter dos Autores
http://twitter.com/lucas_simoes
http://twitter.com/zanonr

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

local
Data Center

Nossos **servidores** dedicados possuem monitoramento 24x7x365, link dedicado e **full duplex**, ou seja, os servidores atingem a **velocidade** do link tanto para **upload** quanto para **download.**

HOST

Servidor
Sites
Gerenciamento

DATACENTER

Co-location
E-mail
Backup

TELECOM

Cross Connection
Banda IP
Link PAP

SOLUÇÕES INTELIGENTES

Solicite um representante comercial

www.localdatacenter.com.br

SOLICITE UM ORÇAMENTO
TEL.: 3956 - 4900
comercial@localdatacenter.com.br

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.